
# An Analysis of the Costs and Benefits of Autocomplete in IDEs

SHAOKANG JIANG and MICHAEL COBLENZ, University of California San Diego, USA

Many IDEs support an autocomplete feature, which may increase developer productivity by reducing typing requirements and by providing convenient access to relevant information. However, to date, there has been no evaluation of the actual benefit of autocomplete to programmers. We conducted a between-subjects experiment (N=32) using an eye tracker to evaluate the costs and benefits of IDE-based autocomplete features to programmers who use an unfamiliar API. Participants who used autocomplete spent significantly less time reading documentation and got significantly higher scores on our post-study API knowledge test, indicating that it helped them learn more about the API. However, autocomplete did not significantly reduce the number of keystrokes required to finish tasks. We conclude that the primary benefit of autocomplete is in providing information, not in reducing time spent typing.

CCS Concepts: • **Software and its engineering** → **Software creation and management**; • **Human-centered computing** → *Empirical studies in HCI*.

Additional Key Words and Phrases: Code Completion, IDE design, API learning

**ACM Reference Format:**
Shaokang Jiang and Michael Coblenz. 2024. An Analysis of the Costs and Benefits of Autocomplete in IDEs. *Proc. ACM Softw. Eng.* 1, FSE, Article 58 (July 2024), 23 pages. https://doi.org/10.1145/3660765

## 1 INTRODUCTION

Autocomplete is a feature in integrated development environments (IDEs) that helps programmers write code. When programmers type, autocomplete proposes options in a *suggestion box* panel showing multiple choices of code that the user may want to insert, which may include a variable name, a method call with parameters inserted, type names, etc.; the choice of order may depend on the IDE. The interface (example shown in fig. 1) also provides a *suggestion details* panel showing documentation for the selected suggestion. Programmers can scroll in each panel to read and select the one they want. In doing so, the IDE can reduce the time required for typing and also provide hints regarding appropriate completions that the user may have difficulty coming up with on their own. However, the proposed options can also be inappropriate. In those cases, programmers can waste time reading irrelevant options, or worse, insert incorrect code that they later must debug. Autocomplete is popularly used, and therefore worth understanding in depth; Amann et al. [2] found that programmers typically use autocomplete on 87% of the days they need to code.

Our work focuses on evaluating the benefits of autocomplete for all programmers. Prior work has measured the autocomplete in terms of *acceptance ratio* [3], which is the ratio of the number of times users accepted an autocomplete suggestion to the number of times autocomplete proposed completion options. Hellendoorn et al. [12] found a typical acceptance ratio of 24% and suggested a potential saved typing efforts. However, it is impossible to infer information about the amount of *time* spent on or saved by autocomplete based on them. In our study, we used an eye tracker to

Authors' Contact Information: Shaokang Jiang, shj002@ucsd.edu; Michael Coblenz, mcoblenz@ucsd.edu, University of California San Diego, La Jolla, USA.


ACM 2994-970X/2024/7-ART58
https://doi.org/10.1145/3660765

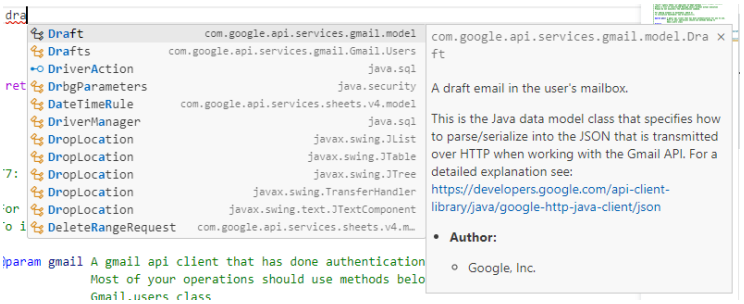



Fig. 1. Autocomplete example, showing the suggestion box (at left) and the suggestion details panel (at right)

study the amount of time participants spent looking at different parts in the IDE, allowing us to differentiate between time spent studying and reading *code* and time spent reading *documentation*.

Even if autocomplete saves programmers time by reducing typing and searching, we hypothesized that it could have a learning impact: perhaps programmers who use autocomplete are less likely to learn key identifiers, such as variable and method names, as well as structural information, such as which methods are supported by classes they rely on, and semantic information about those methods. If true, autocomplete usage could inhibit programmers' abilities to discuss programs with others or to design software systems appropriately. Likewise, if autocomplete inhibits learning, then perhaps students should program without autocomplete.

We conducted a controlled experiment with 32 participants in which some participants were given an IDE with autocomplete disabled (the control group) and some were given an IDE with autocomplete enabled (the treatment group). Next, we asked participants to complete ten programming tasks involving Gmail's Java API, with which the participants were unfamiliar. Then, we gave programmers a quiz so that we could compare API learning outcomes across the two groups. Finally, we collected open-ended responses to survey questions.

We considered several research questions whose answers could deepen our understanding of the tradeoffs of autocomplete. We generated a hypothesis for each:

**RQ1:** Does autocomplete affect how productive developers are?
**Hypothesis 1:** Programmers who use autocomplete complete more programming tasks within the fixed duration of the study.

**RQ2:** How does autocomplete usage affect how much developers learn about APIs?
**Hypothesis 2:** Using autocomplete inhibits API learning.

**RQ3:** How does autocomplete usage affect the time spent reading the documentation?
**Hypothesis 3:** Autocomplete reduces the time spent in reading documentation.

**RQ4:** How does autocomplete usage affect the number of keystrokes programmers enter?
**Hypothesis 4:** Autocomplete reduces the number of keystrokes required to complete tasks.

We also considered four exploratory research questions, which do not have associated hypotheses:

**RQ5:** Wang et al. [35] found that the benefit of autocompleted parameters had not been studied. We ask: what is the performance of autocompleted parameters?

**RQ6:** How do the benefits from native speaker status or programming experience compare with the benefits of autocomplete?

**RQ7:** When IDE designers create autocomplete features, they need to choose how many options to show or prioritize. What is the distribution of gaze points over positions in the autocomplete *suggestion box*?

**RQ8:** What is the relationship between typing speed and programming experience or touch typist status?

Table 1. Research questions and metrics used for assessment

| Research Questions | Metrics for assessment |
|---|---|
| RQ1: Productivity | M1: Number of tasks completed<br>M2: Time spent in finishing each individual task |
| RQ2: Learning | M1: Quiz scores<br>M2: Survey responses |
| RQ3: Documentation reading time | M1: Time spent on each area |
| RQ4: Saved typing efforts | M1: Number of keystrokes while programming<br>M2: Measured typing speed (section 3.2.2)<br>M3: Autocomplete usage info (section 3.2.5) |
| RQ5: Parameter autocomplete usage | M1: Autocomplete usage info (section 3.2.5)<br>M2: Survey responses |
| RQ6: Influence of native language and programming experience | M1: Quiz score<br>M2: Number of tasks completed<br>M3: Time spent reading documentation |
| RQ7: Patterns of autocomplete box usage | M1: Raw gaze point distribution<br>M2: Autocomplete usage info (section 3.2.5) |
| RQ8: Typing speed's relation to programming and touch typing skills. | M1: Measured typing speed (section 3.2.2) |

For each research question, we measure and analyze the data using the metrics mentioned in table 1. At the end of the results section, we present participants' comments on the autocomplete system in our post-study questionnaire.

*Contributions.* Surprisingly, we found that although autocomplete reduced keystrokes in almost all individual accepted autocomplete sessions (86%), autocomplete did not reduce the total number of keystrokes required to complete tasks. However, autocomplete did significantly reduce the completion time. The mechanism of autocomplete's success in helping programmers complete tasks was that it reduced the time developers spent reading documentation.

We had hypothesized that autocomplete usage might interfere with learning, since it might inhibit reading and understanding important sections of the documentation. In fact, in addition to helping developers complete tasks faster, autocomplete led to a significant *increase* in learning as measured by our post-study quiz. We conclude that autocomplete should be used even in educational contexts in which learning is the primary goal in addition to contexts in which productivity is the main objective.

*Implications.* Our focus is on traditional autocomplete for Java programming, but we expect that our results generalize to other languages in which high-fidelity autocomplete options can be provided. As AI-based autocomplete tools, such as Copilot, become more popular, it will be important to re-evaluate the learning implications, since these tools may reduce the cognitive involvement of programmers. However, these tools may also serve as advanced search engines in the same way as autocomplete, enabling quick access to common patterns of API use. The results may also apply to autocomplete in other scenarios, such as search bar suggestions, which may actually serve as tiny search tools on their own.

## 2 RELATED WORK

*Autocomplete.* Amlekar et al. [3] compared autocomplete usage across experienced and novice software engineers by analyzing the autocomplete events from 35 volunteers. They found no difference between the two groups and a maximum average autocomplete acceptance ratio of around 25%. Hellendoorn et al. [12] also found a similar acceptance ratio of 24%. They also investigated situations where programmers disagreed with suggestions and the performance of existing autocomplete models. Our study focuses instead on the overall benefit of autocomplete. Also, unlike Hellendoorn et al.'s paper, our study is a controlled study, comparing autocomplete to non-autocomplete.

Wang et al. assessed comments from interviews with experienced programmers [35], finding that programmers expect to use parameter autocomplete features. Our focus is on a quantitative study evaluating the potential benefits of autocomplete.

Various studies [6, 20, 27, 31, 36] propose improvements to autocomplete techniques, such as designing new algorithms and using better datasets, to make autocomplete work more effectively. However, in some cases they lacked data regarding whether and in which situation users benefit from autocomplete. Our approach is to analyze in what ways autocomplete benefits programmers.

*AI-assisted autocomplete.* Studies have found conflicting results regarding whether GitHub Copilot reduces task completion times. Peng et. al. [24] compared the productivity effect of Copilot to a standard IDE with 95 professional programmers and found that the Copilot group completed tasks 55.8% faster; people with less programming experience benefited the most. In contrast, Vaithilingam et al. [32] conducted a study with 24 programmers and 20-minute tasks, finding that Copilot did not improve task completion time. One possible explanation for the discrepancy is that Copilot helps with some kinds of tasks but not with others. *Perceptions* of productivity may also matter; Ziegler et. al. [37] found that the acceptance ratio of copilot suggestions and the number of accepted characters affect how helpful developers believe Copilot is. Our work measures task completion time, documentation reading time, learning, and total keystrokes required, providing a more comprehensive perspective on the impact of completion techniques.

Other studies considered typing burden. Imai found that Copilot reduces programmers' typing burden but inserts superfluous code that must later be removed manually [15]. Liang et al. [21] also found that Copilot reduces keystrokes but that participants found it potentially annoying and unhelpful in brainstorming potential solutions.

Autocomplete can also be useful in non-programming contexts, such as in messaging completion [22], image completion [19] and data exploration completion [29]. Separate evaluations are appropriate for autocomplete systems in other contexts.

*Experimental techniques.* Limitations in existing eye tracking software led us to develop our own software. Some existing approaches require artificial limitations while coding [1, 9, 30]; others require setting up a fixed area of interest (AOI) during the experiment [7, 18]. Still others are built on Visual Studio, which is not as popular as VSCode [34] and do not provide a convenient facility for monitoring the areas of views of interest [23, 26]. Finally, our initial testing found that some existing tools, such as iTrace [11] had high latency on the hardware we had available. Our monitoring environment is built on top of VSCode. Unlike other work on API usability [8], which used window focus to monitor participants' time spent on reading documentation and coding, our approach uses an eye tracker with a split window to detect the region the user is reading. Our method may reduce the influence of requiring keyboard commands to switch windows while coding.

## 3 METHOD

We conducted a between-subjects experiment in VSCode (Version 1.80.0). In the control condition, participants used VSCode with autocomplete disabled; participants in the treatment condition used standard autocomplete of VSCode (a.k.a IntelliSense) supported by Language Support for Java(TM) by Red Hat extension (version 1.20.0). We neither encouraged nor discouraged the autocomplete group to use autocomplete. During the experiment, participants were expected to learn how to get information from Gmail using Gmail API for Java. We also used an eye tracker to monitor the region participants were focusing on for the coding part (part 1), as well as the line number that they were looking at inside the autocomplete suggestion box. A complete reproduction package with detailed step-by-step instructions is included in the paper supplement. The study contained four parts as follows.

*Part 0: General instructions (15 minutes).* Participants were required to read a brief introduction to the experiment and a basic introduction to Gmail API for Java. They were allowed to visit any part of the entire documentation but only four out of 32 participants elected to read beyond the introduction[1]. We explained that participants could ask questions until part 1 started.

*Part 1: Coding (110 minutes).* We gave 10 coding tasks, each consisting of one method that needed to be completed. Participants had access to documentation and could execute the provided test cases by clicking the *check* button above each task, using the VSCode command or using a menu button. After clicking *check*, the result was shown within 20 seconds. Participants were required to run the tests when finishing each task. During development, participants could utilize a faster approach by using the `main` method in Java, with prefilled test cases provided by us for debugging, returning the result nearly instantly. To complete part 1, participants needed to either finish all ten tasks or run out of time. Participants were allowed to return to earlier tasks. We asked participants to always look at the screen[2], but participants were permitted to look at the keyboard or take breaks as needed. We did not answer questions about the tasks except for very basic syntax questions, such as how to write an assignment statement; and questions about finding top-level components of the documentation, defined as any header link in the introduction to the experiment page, such as where is the entry to Gmail API's Javadoc.

*Part 2: Quiz (no time limit).* Participants were required to finish a 24-question quiz that assessed their knowledge of the Gmail API.

*Part 3: Post-study survey (no time limit).* Participants were required to complete a survey containing questions related to demographics and autocompletion usage.

*Participants and recruitment.* We recruited 32 programmers via posters on campus and via interactions with computer science students on campus. Thirty participants self-identified as 1) having at least one year of experience with Java 2) having no previous experience with Gmail API for Java and 3) at least 18 years old. Two participants self-identified in categories 2) and 3), with at least one year of programming experience and having learned Java over at least half of a year recently. Our study was approved by the Institutional Review Board (IRB) of the University of California, San Diego, and all programmers signed a consent form before starting.

[1]Two participants misunderstood our instructions and didn't finish reading the instructions until they formally started part 1; we granted five extra minutes for them to read the instructions before they started part 1.
[2]At the time, we understood this to be important for gathering consistent data; in retrospect, it does not appear to have been necessary.

### 3.1 Experiment description

*Environment.* The study took place on a laptop computer (Dell Inspiron 15 7567 with Intel i5-7300HQ) running Windows 10 ver. 19045 with an external keyboard (Dell L100), mouse (Logi M170) and 25" monitor (Acer G257HU) with 2048*1152 resolution. A Tobii Eye Tracker 5, a camera-based eye tracker using both eyes to determine gaze position, was attached at the center bottom of the monitor to collect eye movement data. We obtained a sampling rate of around 50 Hz containing filtered gaze data defined by Tobii. Since we did not use a separate fixation algorithm, we refer to this data as *raw gaze* in this paper. According to Housholder et al. [14], the typical accuracy for the tracker is 1.01°. To provide a comfortable and realistic environment, we did not use a chin rest.

Each participant was required to adjust the seat, calibrate the tracker, align with the center of the screen, and keep an estimated 55 ± 5 centimeters between the screen and their head before starting. We used Tobii Experience to calibrate the tracker and only recalibrated it if the participant left the seat. Then, the participant started the experiment with a pre-configured web-based VSCode instance to finish parts 0 & 1, and used Zoom to record the entire screen for further analysis. After finishing coding tasks, they would be redirected to Qualtrics to finish the rest parts.

To get enough participants while maximizing the possible benefits of autocomplete usage, we selected Java, which is a widely used statically-typed programming language. We asked participants to not use copy and paste to ensure that participants would not merely copy and paste code from the documentation or their previous code. Finally, due to limitations in our eye-tracking software, participants could not change the panel organization of the left panel (coding) or the right panel (documentation) and were required to use the keyboard when interacting with autocomplete suggestions instead of selecting an autocomplete item via mouse click[3].

To improve external validity and maximize the chances of the non-autocomplete group (which relied on the quality of the documentation), we wanted to base our tasks on a mature API with high-quality documentation. We selected the Gmail API for Java and used its existing documentation from Google.

*Documentation.* The right panel of the screen showed documentation (fig. 2), consisting of a locally hosted website that provided entries to 1) today's agenda; 2) a page we wrote that introduced how to use the Gmail API, including one sample usage with an explanation; 3) documentation for `ArrayList`, `Hashtable`, and Gmail API from official sites; 4) an official Gmail API introduction and usage samples provided by Google.

*Coding Tasks.* We designed tasks according to the following criteria:

- Representative of realistic usage of the Gmail API, such as sending email
- Maximize the possibility of using autocomplete based on previous research by Amlekar et. al [3] and Hellendoorn et al.[12]
- Ranged in difficulty from simple to complex based on our estimate of difficulty.

We started with one warm-up task (T0) to help participants get familiar with the environment. In the first and second tasks (T1 and T2), participants needed to build a request using the Gmail API. The solution code for the first task was provided in the documentation. In T3–T7, participants learned different ways to build, send, and use the returned value from the API. For T8–T9, participants needed to use the API to finish some advanced operations, such as getting and filtering recent emails. Tasks after T0 came in pairs so that participants would first use an unfamiliar API and then use that API in some exercises.

[3]Among 13 participants whose full autocomplete data were analyzed, seven still used mouse clicks to select an autocomplete item, resulting in a total of 14 autocomplete sessions. These sessions accounted for only 1.5% of the 917 total autocomplete sessions they produced. We excluded this data from any analysis related to research questions.

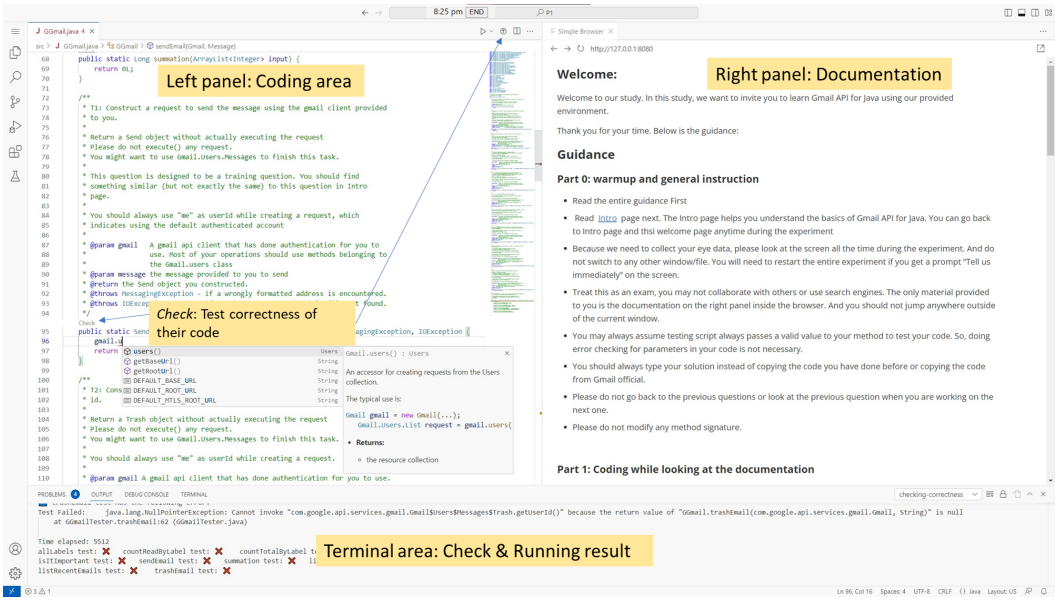



Fig. 2. A sample running environment for programming tasks

*Quiz.* After part 2, we issued a 24-question quiz, graded on a 53-point scale (some questions had multiple components) to assess the knowledge each participant had acquired. Questions fell into three categories, which were not revealed to participants:

- *Understanding* questions (19 points): Understanding the structure of the Gmail API, including the class hierarchy.
- *Memorizing* questions (17 points): Remember the method names and detailed content each class provides.
- *Exploration* questions (17 points): Exploring material that is not strictly required for the tasks to assess what knowledge participants had acquired incidentally while reading or browsing documentation.

Questions were separated into multiple pages and participants could only move forward.

*Post-study survey.* To collect demographic data and subjective comments about autocompletion, participants finished a post-study survey after finishing the quiz.

### 3.2 Data collection and pre-analysis

We conducted the experiment with the web version of VSCode with Extension Pack for Java extension installed to facilitate our development of analysis software. We created a Chrome extension that captured relevant events in the IDE and communicated with a C#-based server process we created, which communicated with the eye tracker and recorded data, using Chrome's *native messaging host* feature[4]. We found this approach to be more reliable and provide lower latency than using a network-based approach. Participants used a custom extension we created for VSCode that added a *check* button, which participants used to run test cases that we provided. In the following subsections, we describe how we computed per-task development time, typing speed, time spent on each area, eye validity data, autocomplete usage info, and eye gaze distribution over the autocomplete box.

[4]https://developer.chrome.com/docs/extensions/mv3/nativeMessaging/

*3.2.1 Per-task development time.* We analyzed screen recordings to compute task times. We measured from the moment the participant's cursor moved into the task area, which we counted as the start time, until the participant correctly completed the task and clicked *check*, which we counted as the end time. All tasks were situated in one file, so because our recording mechanism created only one recording per participant for all tasks, and participants were permitted to navigate freely among tasks, we needed to compute task times manually. By determining these start and end times manually, we also excluded time spent *after* correct completion, e.g., if participants refactored their code after completing a task.

For participants who returned to earlier tasks rather than completing all tasks in order, the time is the sum of the times of all sessions in which participants worked on the task. As with the first session, each later session of a task was considered to start when the mouse moved into the task area and the code panel did not scroll for five seconds. Each session ended when the mouse left the area and the code panel did not scroll for five seconds. We did not consider the corresponding session to end in the following scenarios: 1) removing lines in another task, which made the file fail to compile; 2) scrolling, not staying in any task for five seconds, and returning to the original task afterward.

*3.2.2 Typing speed.* To estimate participants' typing speed, one quiz question asked participants to type a line of code as fast as they could. We embedded JavaScript code into the quiz implementation on Qualtrics to monitor the time participants spent typing, defined as the interval from when participants started typing to when they typed the last character. If the input box loses focus and then regains it, such as when participants click somewhere outside the input box and then return to typing, another typing session would be initiated. The final total typing time is the summation of all sessions. We calculated typing speed by dividing the total number of characters (at least 120) by the number of milliseconds spent, thus including time imposed by fixing typing errors.

*3.2.3 Time spent on each area.* To analyze how autocomplete affects the time spent reading documentation (RQ3), we estimated the time participants spent looking at each area on the screen: the documentation area, the coding area, and the terminal area.

The VSCode window was split horizontally into parts for coding and documentation, with the coding panel occupying 52% of the width on the left. A resizable terminal component was available below. When autocomplete appeared (as in fig. 2), its interface was opaque and obscured other components. Our Chrome extension tracked the terminal view's size and the position of the autocomplete popups while keeping the server updated.

To estimate the time spent on each area, our software recorded the categorized position data on which participants focused: left panel, right panel, or terminal area. The autocomplete view is opaque and, when it appears, can obscure part of all three areas. We attributed any raw gaze that fell on autocomplete components to the left panel because autocomplete only appears while programming. To estimate the duration, we first built a series of sessions, where each session contains consecutive valid eye data points in one area. Then, we removed any sessions that were shorter than 225 ms, based on prior work showing that users need an average of at least 225 ms to formulate a meaningful reading event under silent reading mode [28]. Then, the total duration in each area is the sum of the durations of all valid sessions corresponding to each area.

*Estimated time spent looking at autocomplete elements.* For analytic purposes, we wanted to consider time spent looking at autocomplete as time spent reading documentation, even though we had previously included it in coding time. Estimated time spent looking at autocomplete elements includes time spent on looking at the *suggestion box* and *suggestion details*. Both are defined as the time the user's raw gaze leaves each area minus the time the user's raw gaze enters the area. As

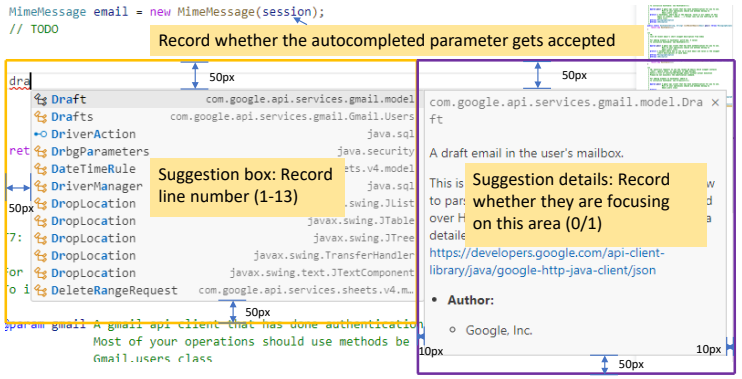



Fig. 3. AOI boundary illustration

recommended by Goldberg et al. and Holmqvist et al. [10, 13], we determined the area with two changeable areas of interest (AOI) formulated as shown in fig. 3, considering the possible vertical drift, the area in which autocompletes normally show up, and the eye tracker performance that we observed. Figure 3 shows a boundary example.

We first built a series of sessions, where each session contains consecutive valid raw gaze points in each area. Prior work [16] showed that even short glances might be useful for acquiring information, so in contrast with estimating time spent in the three major areas (left, right, terminal), we wanted to consider even short glances at autocomplete. Therefore, we summed up all autocomplete gaze sessions over all areas because the minimal duration of a glance is less than 100 ms [16, 33].

*Documentation reading time.* We define the duration of focusing on the documentation area as the time from when the user's raw eye gaze enters the documentation area to when it leaves the documentation area, with a minimum duration of 225 ms [28]. The time also includes the estimated time spent looking at autocomplete elements, considering that the content from any autocomplete components also serves as a form of documentation. When calculating the estimation, it is the summation of time spent on the right panel and the estimated time spent looking at autocomplete elements.

*Coding time.* We define the duration of focusing on the coding area as the time from when the user's raw eye gaze enters the coding area to when it leaves the coding area, with a minimum duration of 225 ms. When calculating the estimation, we subtract the estimated time spent looking at autocomplete elements from the time spent on the left panel.

*Terminal time.* We define the duration of focusing on the terminal area as the time from when the user's raw eye gaze enters the terminal area to when it leaves the terminal area, with a minimum duration of 225 ms.

*3.2.4 Eye validity data.* We collected eye data validity information so we could analyze how effective our eye tracker was. We computed *eye validity ratio*, the number of valid gaze points divided by the total gaze points; both sets of data were provided by the eye tracker.

*3.2.5 Autocomplete usage info.* To find evidence regarding RQ4, RQ5, & RQ7, our Chrome extension recorded each autocomplete session for the group that used autocomplete. To ensure correctness, we extracted *Final content* and *Autocompleted parameter usage* from each recorded session manually afterward.

- *Final content*: The final content the user typed or selected (extracted manually).

- *Autocompleted parameter usage*: Whether the user modified the autocompleted parameter after acceptance (extracted manually).
- *Acceptance status*: Whether the user accepted the autocomplete session with the *enter* or *tab* keys (captured automatically).

*3.2.6 Eye gaze point distribution over the autocomplete box.* To analyze the gaze distribution over the autocomplete *suggestion box* (RQ7), we collected the y-axis offset within the suggestion box for each gaze point that fell within the box.

## 4 RESULTS

We randomly assigned participants to conditions using block randomization with size two. Among the 32 participants, four (two from each condition) were excluded from the analysis except for their post-study survey results because they completed fewer than four tasks, indicating substantially weaker programming skills.

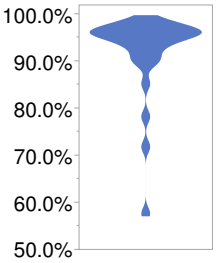


Fig. 4. Eye validity for all participants

We analyzed the data with a collection of JavaScript and Java scripts (provided in the paper supplement), Excel, and JMP Pro 17.1. To confirm that both groups had similar kinds of participants, we first analyzed the distribution of previous experience. A Wilcoxon test does not show a significant difference in coding experience ($p \approx 0.63$) between the autocomplete group ($M \approx 6.6, SD \approx 4.6$, median = 5) and the group without autocomplete ($M \approx 5.3, SD \approx 2.1$, median = 5). A chi-squared test also does not show a significant difference between the two groups ($p \approx 0.25$) regarding the use of English as a native language. In the autocomplete group, 5 out of 14 participants reported English as their native language. In the without autocomplete group, 8 out of 14 participants reported English as their native language. All participants are or used to be students in a US university. Among all participants, a chi-squared test shows no significant difference ($p \approx 0.49$) in degree and professional status between the autocomplete group (4 MS, 2 Ph.D., and 8 undergraduate) and without autocomplete group (6 MS, 1 Ph.D., 1 software engineer, and 6 undergraduate). We also compared eye validity measurements and did not find a significant difference across conditions ($p \approx 0.91$) using ANOVA. Figure 4 shows the distribution of eye validity; for 23 participants (82%), eye validity was at least 90%. In addition, we found an autocomplete acceptance ratio of 30%, which is comparable to that observed in prior studies [3, 12].

### 4.1 RQ1: Does autocomplete affect how productive developers are?

**Hypothesis 1:** Programmers who use autocomplete complete more programming tasks within the fixed duration of the study.

If autocomplete helps programmers complete tasks in less time, then given the fixed total time of our study, it should enable participants to complete more tasks. Figure 6 shows a violin plot of the number of tasks participants finished in each condition. A Wilcoxon Two-Sample test does not show a significant effect of autocomplete on the number of tasks finished ($p \approx 0.11$).

Because we found an insignificant result of the number of tasks finished across conditions, we also considered the time participants spent on each task as defined in section 3.2. Figure 5 shows a violin plot of the task time distributions. Due to nonlinearity of completion times, we log-transformed the times before analysis. Using a Restricted Maximum Likelihood (REML) approach with a fixed effect for completion time and a random effect for the task, we found a significant difference

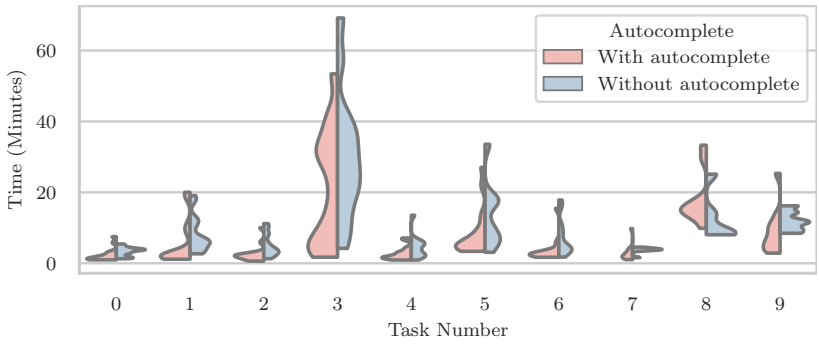



Fig. 5. Per-task time vs autocomplete usage

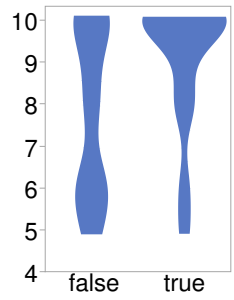



Fig. 6. Finished tasks vs autocomplete

($p < 0.0001$) across conditions with $R^2 \approx 0.54$. A parameter estimation shows participants who used autocomplete are 8.2% faster than the group without autocomplete.

### 4.2 RQ2: How does autocomplete usage affect how much developers learn about APIs?

**Hypothesis 2:** Using autocomplete inhibits API learning.

When proposing completions, the autocomplete user interface only shows part of the documentation. We hypothesized that autocomplete could inhibit learning for its users compared with users who only have access to the regular documentation interface.

To test this hypothesis, we gave participants a quiz after they finished the tasks. We first analyzed the relationship between autocomplete usage and score [$F(1, 26) \approx 8.3, p \approx 0.0079, d = 1.1$] using ANOVA. Figure 7 shows the result. Although we found a significant effect of autocomplete on learning, its direction was opposite our prediction: in fact, autocomplete *promotes* learning. On average, participants who used autocomplete ($M \approx 38, SD \approx 1.3$) got 5.4 points more than the group without autocomplete ($M \approx 32, SD \approx 1.3$).

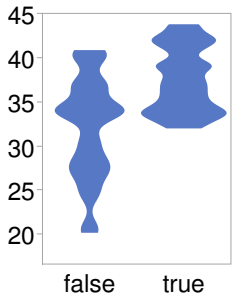



Fig. 7. Quiz Score vs autocomplete usage

As discussed in section 3.1, all questions were designed to belong to one of three categories, 1) *Understanding* questions, 2) *Memorizing* questions, and 3) *Exploration* questions. We further analyzed the relationship between autocomplete usage and score in each category.

We found a significant relationship between autocomplete and performance on *understanding* questions ($p \approx 0.024$). Because the scores for *understanding* questions were not normally distributed, we used the Wilcoxon Two-Sample Test to analyze them. Participants who used autocomplete ($M \approx 14, SD \approx 2.2$) averaged 2.3 more points than the group without autocomplete ($M \approx 12, SD \approx 3.0$).

An ANOVA test on *exploration* questions gave a significant result [$F(1, 26) \approx 6.0, p \approx 0.021, d = 0.93$]. On average, participants who used autocomplete ($M \approx 11, SD \approx 2.0$) got 2.2 points more than the group without autocomplete ($M \approx 8.6, SD \approx 2.7$).

An ANOVA test did not show a significant effect of autocomplete on *memorizing* questions [$F(1, 26) \approx 1.7, p \approx 0.20, d = 0.50$]. On average, participants who used autocomplete ($M \approx 13, SD \approx 1.8$) got 0.91 points more than the group without autocomplete ($M \approx 12, SD \approx 1.9$).

We also asked participants for their opinions on whether autocomplete influences learning in general. We manually classified each free-response answer as *promoting learning*, *inhibiting learn-*

*ing*, or being *unsure*. Table 2 shows the result, which was consistent with our findings from the quiz: 59% of participants felt autocomplete promotes learning.

Table 2. Beliefs about how autocomplete influences learning

| Inhibits learning | No effect | Promotes learning | Unsure |
|---|---|---|---|
| 5 (16%) | 3 ( 9.4% ) | 19 (59% ) | 3 (9.4% ) |

We conducted open coding on the responses. Fifteen participants suggested autocomplete helps explore and remember the content. Top comments included helping users to find, learn and apply the right method/attributes, reducing the need to read the documentation. Twelve participants suggested autocomplete is good for learning unfamiliar content, emphasizing the benefits of showing the type of each variable. Nine participants felt that autocomplete may inhibit learning by making it impossible to understand APIs extensively and inhibiting thinking. One participant suggested "[Autocomplete] lessens the need for users to learn API in detail since the user can kind of guess and let the autocomplete tool do the job of completing the method call."

*Discussion.* We were surprised that autocomplete promotes learning instead of inhibiting learning. Autocomplete's benefit in learning is likely because it provides convenient search: it includes only APIs that match the prefix the user entered. Although the *suggestion details* box does not provide a complete perspective, overall it seems to be more effective in terms of time.

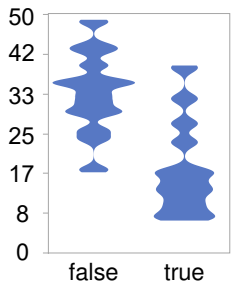
The overall quiz score is consistent with the survey response showing autocomplete is helpful in learning. The quiz score for *Exploration* questions is somehow consistent with RQ1 suggesting autocomplete could help programmers explore more functionality in a limited time, thus learning more efficiently. The quiz score for *Understanding* questions shows autocomplete helps programmers understand the structure of a library. It is not a surprise that no significant differences in quiz scores for *Memorizing* questions are observed because *Memorizing* questions are in some sense assessing exactly what participants did.

### 4.3 RQ3: How does autocomplete usage affect the time spent reading the documentation?

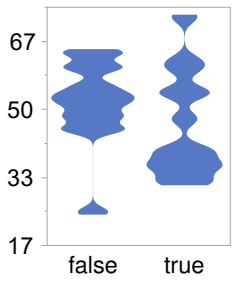
**Hypothesis 3:** Autocomplete reduces the time spent in reading documentation.

We estimated the time participants spent reading documentation as described in section 3.2, which includes any duration of raw gaze that falls on autocomplete components, considering autocomplete is a form of documentation.

We found that usage of autocomplete significantly reduced time spent reading documentation $[F(1, 25) \approx 21, p \approx 0.00012, d = 1.76]$ using ANOVA. Figure 8(a) shows the distribution. On average, autocomplete users ($M \approx$ 18min., $SD \approx$ 2.5min.) spent 16 fewer minutes reading documentation than those in the control condition ($M \approx$ 34min., $SD \approx$ 2.4min.).




(a) Reading duration (min.) vs. autocomplete




(b) Coding duration (min.) vs. autocomplete

Fig. 8. Reading and coding durations

*Coding time.* We also analyzed whether autocomplete affected time spent on *other* areas, first by considering the code region. The estimated coding time excludes any estimated time spent looking at autocomplete elements, as described in section 3.2 We did not find a significant effect of autocomplete on the coding duration [$F(1, 25) \approx 1.6, p \approx 0.22, d = 0.48$] using ANOVA. The distribution is shown in fig. 8(b). Comparing means, the autocomplete group ($M \approx$ 46min., $SD \approx$ 3.2min.) spent 4.9 minutes less than the non-autocomplete group ($M \approx$ 52min., $SD \approx$ 3.1min.).

*Terminal time.* We did not find a significant effect of autocomplete on total terminal duration [$F(1, 26) \approx 0.055, p \approx 0.82, d = 0.089$] using ANOVA. Comparing means, the autocomplete group ($M \approx$ 5.8min., $SD \approx$ 0.72min.) spent 14 fewer seconds ($M \approx$ 6.0min., $SD \approx$ 0.72min.).

*Limitations.* Our time estimation relies on the duration of raw gaze points without fixations, which could lead to inaccuracies in determining the actual duration people spend on each area of the screen. A more advanced eye tracker with the appropriate capabilities could yield better results and deepen our understanding of the cognitive behavior behind it.

Our method of estimating documentation reading and coding times (in which we considered even glances at autocomplete to represent documentation reading) could have introduced a bias toward longer documentation reading times for autocomplete users. However, since we found that autocomplete users spent *less* time reading documentation than non-autocomplete users, the effect of any bias does not appear to have been significant. Our approach may also have underestimated the time spent coding for autocomplete users. However, we believe that any effect of this was negligible compared to the difference between means (4.9 minutes).

### 4.4 RQ4: How does autocomplete usage affect the number of keystrokes programmers enter?

**Hypothesis 4:** Autocomplete reduces the number of keystrokes required to complete tasks.

We analyzed this hypothesis in three ways. First, we considered the number of keystrokes saved in each accepted autocomplete session compared to typing the text that autocomplete inserted. Second, we estimated how much typing time autocomplete saved in each session by considering each participant's typing speed collected from the quiz with monitoring tools as described in section 3.2. Finally, we compared the total number of keystrokes across conditions.

For each autocomplete session that a participant accepted (via *enter* or *tab*), we manually extracted the final text entered by autocomplete, which included any automated parameter suggestions regardless of participants' final acceptance, to obtain a length in characters ($M = 9.1, SD = 4.3$). We also collected any keystrokes that happened during each autocompletion session. We computed saved keystrokes as the final word length minus the number of typed characters. Autocomplete can also *cost* typing, resulting in negative values. The distribution of the result is shown in fig. 9(a); autocomplete most often saves three to six characters. Autocomplete reduced keystrokes in 1254 sessions (86%).

We then estimated how much typing time autocomplete saved. We multiplied each participant's code typing speed (section 3.2.2) by the number of characters they avoided typing. Then, we subtracted the estimated time spent looking at autocomplete elements. The distribution of the result is shown in fig. 9(b). Typically, when autocomplete saved time, it saved less than two seconds. Overall, autocomplete reduced time in 1147 sessions (79%).

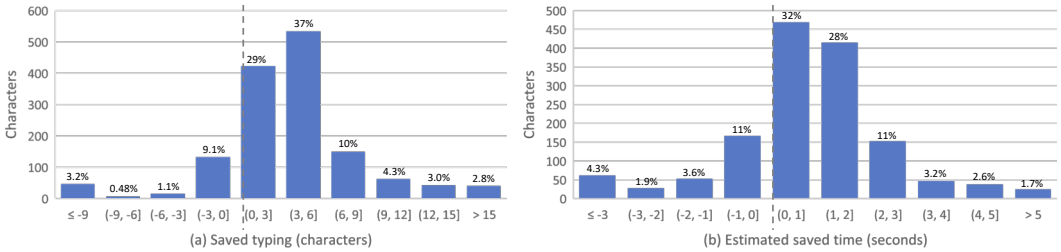



Fig. 9. (a) distribution of the number of characters saved (b) distribution of estimated time saved

Comparing the total number of keystrokes across experimental conditions (shown in fig. 10), we did not observe a significant effect of autocomplete [$F(1, 25) \approx 0.22, p \approx 0.65, d = 0.18$] on the number of keystrokes using ANOVA.

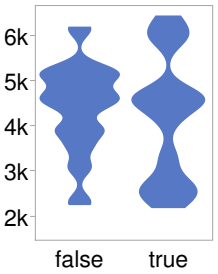



Fig. 10. Keystrokes vs. autocomplete usage

*Discussion.* Our finding that autocomplete did not reduce the number of keystrokes contradicts previous research [12] that suggested autocomplete would reduce keystrokes. This also appears to contrast with fig. 9, which shows that 85% of the participants appear to have saved typing. There are two possible reasons: 1) we did not monitor programmer behavior after they accepted suggestions, so they may eventually have undone the insertion; 2) programmers might waste keystrokes exploring options in rejected autocomplete sessions. In general, acceptance can result in additional work for programmers, such as removing parameter autocomplete content. Hence, we believe comparing total task keystrokes is a more robust approach than considering sessions individually.

We initially considered the possibility that some participants did not use autocomplete enough for us to observe an effect, so we considered the correlation of the number of autocompletion sessions, personal preference for autocomplete, and the coding time with the number of keystrokes [$F(3, 8) \approx 11, p \approx 0.0037$] using ANOVA (we included the coding time because longer task time may suggest more typing required, regardless of autocomplete). The number of autocomplete sessions had a significant positive relationship ($p < 0.001, \eta^2 \approx 0.70$) with the number of keystrokes, but the coding time ($p \approx 0.38, \eta^2 \approx 0.022$) and personal preference ($p \approx 0.99, \eta^2 \approx 0$) did not: more autocompletion events correlated with more keystrokes.

If autocomplete helps improve productivity, it appears that reducing time typing is not the primary mechanism when learning new API. Instead, our interpretation is that when autocomplete reduces task time, it does so by reducing *reading* time. This corresponds with our results for RQ3 regarding whether autocomplete reduces reading time.

Because our analysis subtracts the estimated time spent looking at autocomplete elements, our estimated saved time could be inaccurate for users who type while looking at autocomplete. However, we expect these users will just type instead of accepting autocomplete's suggestion, so we do not believe these cases were included in the analysis.

### 4.5 RQ5: What is the performance of the autocomplete for parameters?

When autocompleting a method call, *parameter autocomplete* inserts a suggestion for each parameter. We observed 1457 total instances of parameter autocomplete. We classified each autocomplete session that participants accepted, studying what changes the participant eventually made for each parameter. 64% of accepted autocomplete suggestions were for method calls, in contrast to cases

where parameter autocomplete does not exist, such as autocompleting a local variable. 17% of all accepted autocomplete sessions included parameters.

Among autocomplete sessions in which autocomplete completed parameters, participants accepted *all* parameters 17% of the time; *only some* parameters 20% of the time; and *no* parameters 63% of the time.

*Discussion.* Participants accepted at least some parameters 37% of the time; compared with the 30% acceptance rate for autocomplete in general, parameter autocomplete appears to be at least as worthwhile. However, unlike with normal autocomplete, programmers must accept parameters if they choose an option that includes them. This might suggest parameter autocomplete would require a higher acceptance rate to satisfy users.

Our participants felt that autocompleted parameters were more useful than average (fig. 12). Some participants also appreciated that autocompleted parameters provide hints regarding the parameter type to use, such as `String` or `Integer`.

### 4.6 RQ6: How do the benefits from native speaker status or programming experience compare with the benefits of autocomplete?

Potential benefits from native speaker status or programming experience include enhancing the learning process; increasing the number of tasks finished in a fixed amount of time; and reducing time spent in reading. We were interested in comparing the influence of autocomplete to the influence of programming experience, defined as the number of years in programming, and native English speaker status, defined according to the first language participants learned at birth (since the task involved significant work reading English documentation).

*Learning.* To understand how relevant factors contribute to learning, we constructed a least-squares linear regression model predicting quiz scores from programming experience, native English speaker status, and the use of autocomplete and investigated the influence of those three factors [$F(3, 24) \approx 6.2, p \approx 0.0030$]. A parameter estimation shows each year of programming experience corresponded with 0.65 additional points, a 1.9% improvement relative to the average score of 34.8 points on the quiz ($p \approx 0.028, \eta^2 \approx 0.13$); autocomplete contributed 4.7 points, a 14.5% improvement ($p \approx 0.016, \eta^2 \approx 0.16$); native language of English also contributed 0.85 points, a 2.4% improvement ($p \approx 0.66, \eta^2 \approx 0.0050$). Thus, autocomplete's benefit on quiz scores was similar to 7.2 years of coding experience.

*Number of tasks completed.* Next, to understand what factors related to the number of tasks participants completed, we constructed a least-squares linear regression model predicting the number of tasks finished from programming experience, native English speaker status, and the use of autocomplete [$F(3, 24) \approx 3.4, p \approx 0.033$]. A parameter estimation shows each year of programming experience corresponded with finishing 0.15 more tasks, a 1.8% improvement relative to the average of 8.4 tasks completed ($p \approx 0.17, \eta^2 \approx 0.058$); autocomplete corresponded with finishing 1.2 more tasks, a 15% improvement ($p \approx 0.091, \eta^2 \approx 0.090$); the native language of English corresponded with finishing 1.0 more tasks, a 13% improvement ($p \approx 0.19, \eta^2 \approx 0.053$). Thus, the influence of autocomplete on the number of tasks completed was approximately equivalent to 8.0 years of programming experience.

*Reading time.* Finally, we constructed another least-squares linear regression model predicting the time spent reading from programming experience, native English speaker status, and the use of autocomplete [$F(3, 23) \approx 9.6, p \approx 0.0003$]. Each year of programming experience corresponded with spending 0.97 fewer minutes in reading, a 3.7% improvement relative to the average of 26 minutes ($p \approx 0.076, \eta^2 \approx 0.066$); autocomplete corresponded with spending 15 fewer minutes

reading, a 44% improvement ($p \approx 0.0003, \eta^2 \approx 0.34$); native language of English corresponded with 1.5 fewer minutes reading, a 6.2% improvement ($p \approx 0.68, \eta^2 \approx 0.0034$). Thus, autocomplete's benefit on reading time is similar to 15 years of programming experience. The benefits of using English as the native language were similar to having two years of programming experience.

*Discussion.* Having a native language of English appears to have benefits for programmers. Further study is required to understand the mechanism more fully and identify mitigation approaches. Providing native-language documentation could reduce reading time, which could enable more tasks to be completed. It could also directly improve learning. Another possible explanation is that native English status correlates with a third variable that we have not identified, such as the quality of previous programming experience. We also did not capture any potential effect of having a native language other than English but belonging to the Indo-European language family.

### 4.7 RQ7: What is the distribution of gaze points over positions in the autocomplete suggestion box?

Considering the differences between reading text (for which a more complex solution to map gaze points to lines of text is beneficial) and reading autocomplete options, we used the most basic *attach algorithm* [5], which maps each gaze point to its closest line index based on the raw gaze data collected as described in section 3.2. The result is shown in fig. 11. We excluded any gaze that was vertically outside of the suggestion box area.

We counted all valid raw gaze from all sessions in which the suggestion list contained at least 12 suggestions. The result is shown in fig. 11. Overall, the number of gaze points decreased gradually before line 7, with the first line containing the most gaze points (16%).

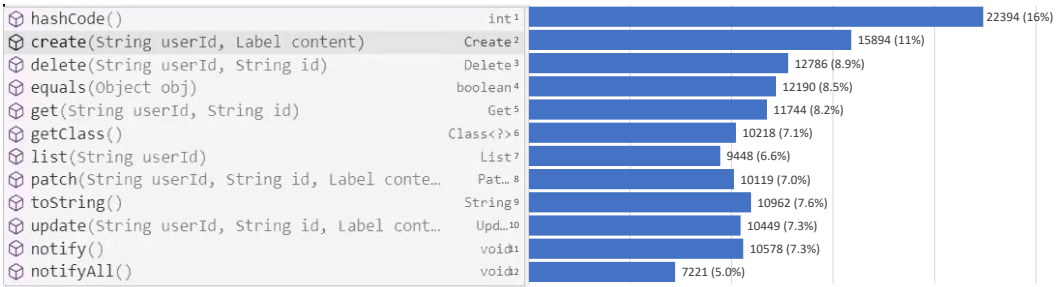



Fig. 11. Distribution of gaze points over autocomplete option indices. The image at left is shown as a sample; the counts at right show total gaze points over the course of the study.

*Discussion.* Because we strictly applied the *attach* algorithm and were limited by the quality of our eye tracker, we excluded a large amount of data ($M \approx 40\%, SD \approx 14\%$, median = 36%). However, the remaining data give a useful picture of how participants read the autocomplete options.

Because our tasks require participants to use unfamiliar APIs, this distribution represents mostly the case when programmers face unfamiliar material, especially for the distribution after line 5. This might not represent the autocomplete focusing distribution in the average case or when participants are more familiar with the content.

### 4.8 RQ8: What is the relationship between typing speed and programming experience or touch typist status?

We estimated participants' typing speed according to the method described in section 3.2.2. An ANOVA test shows programming experience has a significant correlation with typing speed

$[F(1, 26) \approx 10.45, p \approx 0.0033, \eta^2 = 0.29]$. Parameter estimation suggests programmers type 0.176 characters faster per second for every year of programming experience.

Typing speed was not normally distributed, so we used a Wilcoxon two-sample test to analyze the relationship between touch typist status and typing speed, obtaining a $p \approx 0.033$. However, since both tests use typing speed as the dependent variable, a Bonferroni correction is necessary. So, the p-value is only significant if it is less than $0.05/2 = 0.025$. Hence the touch typing relationship with typing speed is not significant but programming experience has a significant correlation with typing speed.

### 4.9 Post-study questionnaire results and discussion

In addition to all quantitative data, we also set up a post-study survey and collected various qualitative data. We conducted open coding on the responses and found a variety of different opinions about the benefits of autocomplete.

*How useful is autocomplete in each of the following situations?* Figure 12 shows participants' opinions on how useful autocomplete is in each context (as defined by VSCode) applicable to Java. Each item is formulated as a 5-point Likert-scale question and includes an example of the context to which the item pertained.

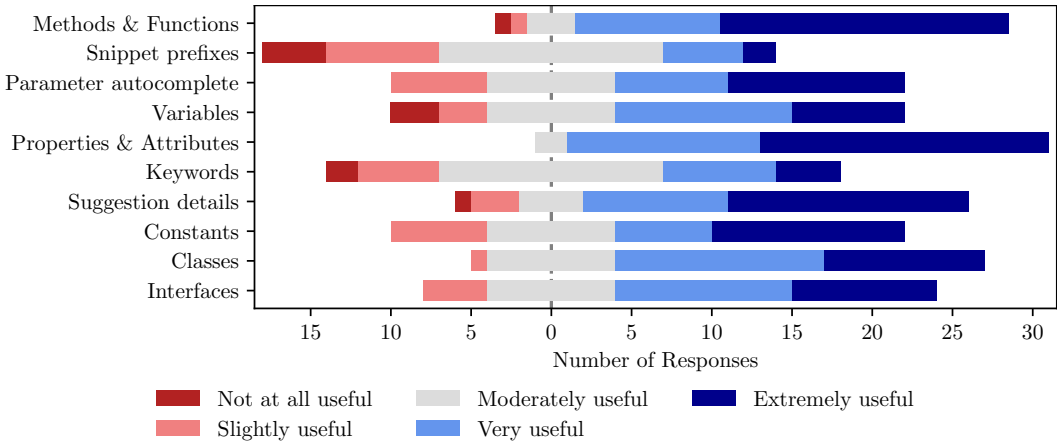



Fig. 12. Responses to: how useful is autocomplete in each of the following situations?

*Snippets* are predefined code templates for language constructs, such as `foreach`. Snippets were rated as the least useful item. Consistent with responses in other parts of the survey, participants reported that properties, attributes, methods, functions, variables, and constants were useful. Some participants suggested autocomplete to be extremely useful for long variable names. Some participants also suggested changing the order of the suggestion content so that user-defined variables and functions always show on the top.

Participants also reported that *suggestion details*, parameter autocomplete, and classes were useful. The positive responses were consistent with our result in the previous hypothesis showing autocomplete reduces reading time and helps in completing parameters.

*In what situations is autocomplete helpful or unhelpful?* We asked two questions about in which situation participants felt autocomplete to be helpful or unhelpful, respectively. Figure 13 shows the result. Most of the participants agreed that autocomplete reduces workload or helps them

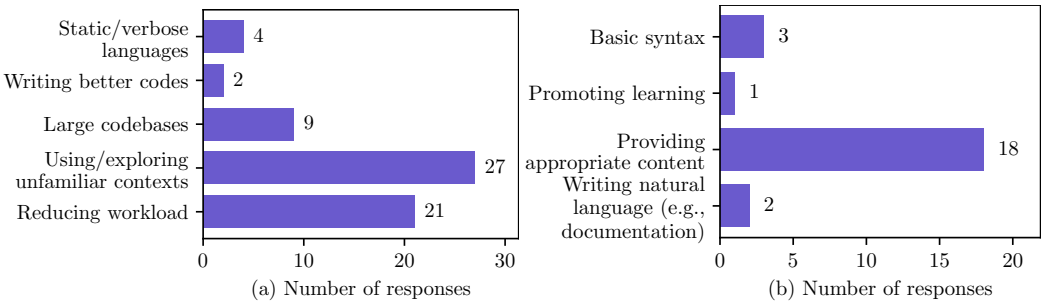



Fig. 13. (a) Autocomplete is helpful in… (b) Autocomplete is unhelpful in…

get familiar with unfamiliar contexts. In terms of reducing workload, most of the participants thought autocomplete reduces the time they need to check the documentation so they can just focus on the logic of finishing each task. Some other participants thought autocomplete would save time or reduce typing effort. Some participants also suggested it helps do repeat work and reduce memorization.

For exploring unfamiliar contexts, most people agreed that autocomplete is useful when using a new library and exploring the name and functionality of each method in it. Some participants thought autocomplete could help them find valid methods but that documentation did a bad job conveying this information. Some participants argued that autocomplete would help them use a new language or switch to another language.

Six (19%) participants thought autocomplete was always helpful and did not provide any comments on the cases where autocomplete is unhelpful. For the other participants, most people thought autocomplete is unhelpful in providing appropriate content. This includes providing incorrect suggestions, providing suggestions in an undesired order, providing too much irrelevant information, and providing inconsistent experience, such as the location or suggestion content, across different IDEs. Eight (25%) participants thought autocomplete might be distracting or annoying due to providing inappropriate content.

Even though we asked them to respond to the questions in general, participants' responses could have been influenced by their experience in our quizzes and coding tasks. This might explain why most people thought it was useful in exploring unfamiliar contexts but not too many thought of writing better code.

*Recommendations for autocomplete features.* We had one additional question asking participants to provide any possible improvements to the current autocomplete tool. Themes we observed are shown in fig. 14.

Among all responses, 5 (16%) participants did not provide any comments suggesting the current autocomplete tool is good enough. In the others, 19 participants suggested multiple ways to improve suggestion content, such as using intelligent suggestions and providing more relevant results. Ten participants suggested the suggested details should include more sample codes or links to online resources. Three participants suggested different ways to invoke the

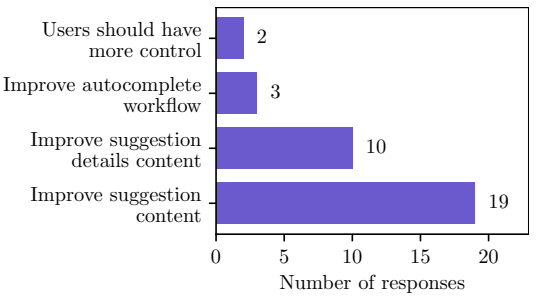



Fig. 14. Recommendations for current autocomplete

autocomplete, such as using a tab, supporting fuzzy search, or the time in providing automatic suggestions, such as showing suggestions while filling in the parameter.

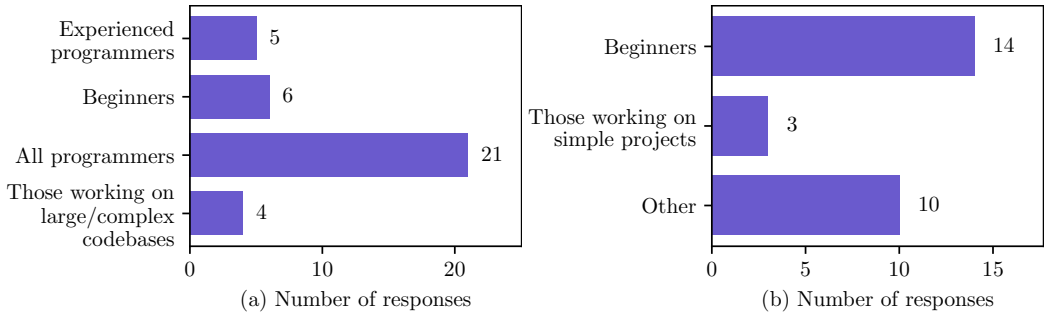



Fig. 15. (a) Who should use autocomplete? (b) Who should NOT use autocomplete?

*Recommend people to use autocomplete or not using autocomplete.* We then have two questions asking participants to recommend and not recommend the type of people to use autocomplete. Results are shown in fig. 15. Participants generally agreed that all programmers who have at least a basic programming background should use autocomplete. Responses also indicated that experienced programmers and those working with complex codebases should use autocomplete more often.

Regarding in what contexts people should use autocomplete (fig. 15(b)), eight (25%) participants suggested everyone should always use autocomplete, thus they were excluded from fig. 15(b). Three participants suggested people who are working on a simple project or the context of the project is simple should not use autocomplete. Ten participants also suggested various situations in which people should not use autocomplete, such as in a certain environment, e.g. exam, on a slow computer, understanding what they need to write before starting, people who just want to guess the correct API, or people who want to memorize everything.

We were surprised about the disagreement regarding whether beginners should use autocomplete: six participants suggested beginners should use it but 14 suggested beginners should not use it. The group that suggested beginners should not use autocomplete focused more on early-stage beginners. Combining this with our results, it would appear that autocomplete is likely beneficial for all but the earliest-stage beginners.

## 5 LIMITATIONS

Our study focused on Java; future work may be needed to assess how the results apply to other languages. Also, our tasks were limited in scope and duration; future studies could investigate the benefits of autocomplete in a longitudinal setting. Our participants were mostly students; it is possible that professionals could obtain different benefits from using autocomplete than students. Finally, our focus was on usage of autocomplete with an unfamiliar API; the implications of autocomplete could be different for APIs with which users are familiar.

We were limited in the amount and precision of data that our eye tracker could capture. We did not compute fixations due to the limitations of our eye tracker (such as the inability to record specific raw points, allowing access only to raw data points). However, a fixation-based analysis might have yielded better data, for instance, for fig. 11. Additionally, our eye tracker's limitations prevented us from conducting a higher-level evaluation of the eye-tracking data, and we did not track edits. Instead, we tracked gaze in relevant screen regions. Tracking across the entire screen

could potentially provide a deeper understanding of programmers' eye movements during coding with autocomplete. In addition, our eye tracker does not provide recalibration functionality or the ability to access the calibration report at any time. A more advanced eye tracker with the ability to recalibrate could yield more accurate results. However, these limitations apply only to RQ7 and, to a lesser extent, RQ3 and RQ6; the remaining results do not rely on eye tracker results.

Our web-based IDE introduced some latency (up to 300 ms) in offering autocomplete suggestions, particularly for participants who typed quickly. A faster autocomplete system could be more beneficial for users. However, we were able to show significant benefits of autocomplete even with this delay, so we think our system's latency was not a significant limitation in our study.

We did not allow participants to use any external resources so that we could collect accurate eye-tracking data for the entire experiment, but two participants expressed that this restriction was unrealistic. However, this restriction was the same between experimental groups, so it did not affect our analysis. Likewise, we disallowed copy/paste, which is unrealistic. However, only one participant complained about this.

Our experiment concerned only one IDE and language; other IDEs and languages may have different autocomplete tradeoffs. However, most IDEs have very similar autocomplete features. Eclipse, like VSCode, shows suggestions as well as documentation. IntelliJ has a similar interface but lacks the suggestion details box. XCode is similar but shows documentation below the suggestions instead of at the right. Support for languages may differ; for example, statically-typed languages generally support better autocomplete than dynamically-typed languages. Further study may be needed for dynamically-typed languages.

## 6 CONCLUSION AND FUTURE WORK

Although it has been widely assumed that autocomplete reduces programmers' keyboard burden, surprisingly, we did not observe this in our study. In contrast, autocomplete enabled programmers to learn an unfamiliar API more successfully and reduced the amount of time they spent reading documentation. We conclude that the primary benefit of autocomplete is providing convenient access to relevant documentation, enabling programmers to learn more effectively.

Some authors expect that more advanced forms of autocomplete, such as Copilot, might inhibit learning [4, 25]. Our finding that autocomplete promotes learning may suggest that more advanced systems could promote learning as well.

AI-based tools, such as Copilot, appear to be the future of autocomplete features. Future work should clearly investigate the learning effects of these more advanced code generation tools. Even so, traditional autocomplete may still be beneficial in certain cases. Proprietary code or APIs with only limited use may not produce a large enough training set to train a language model. Simpler autocomplete features may be less intrusive and less distracting than features that emit very large completion suggestions. Finally, autocomplete may lead a stronger sense of ownership of the resulting code than AI-based code generators, affecting how people feel about the programs they write.

Our study did not consider the order of showing autocomplete suggestions or how many suggestions to show; these aspects could also be investigated in future work. Likewise, further research is needed to understand which attributes of programming assistants could facilitate learning and which might inhibit it.

## 7 DATA AVAILABILITY

A replication package, data from the study, and analysis scripts are available in the ACM Digital Library [17].

## REFERENCES


[1] Maike Ahrens. 2020. Towards Automatic Capturing of Traceability Links by Combining Eye Tracking and Interaction Data. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*. 434–439. https://doi.org/10.1109/RE48521.2020.00064

[2] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. 2016. A Study of Visual Studio Usage in Practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 124–134. https://doi.org/10.1109/SANER.2016.39

[3] Rahul Amlekar, Andrés Felipe Rincón Gamboa, Keheliya Gallaba, and Shane McIntosh. 2018. Do Software Engineers Use Autocompletion Features Differently than Other Developers?. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 86–89. https://doi.org/10.1145/3196398.3196471

[4] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258* (2021).

[5] Jon W Carr, Valentina N Pescuma, Michele Furlan, Maria Ktori, and Davide Crepaldi. 2022. Algorithms for the automated correction of vertical drift in eye-tracking data. *Behavior Research Methods* 54, 1 (2022), 287–310.

[6] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An Empirical Study on the Usage of BERT Models for Code Completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 108–119. https://doi.org/10.1109/MSR52588.2021.00024

[7] Daniel Kyle Davis and Feng Zhu. 2022. Analysis of software developers' coding behavior: A survey of visualization analysis techniques using eye trackers. *Computers in Human Behavior Reports* 7 (2022), 100213. https://doi.org/10.1016/j.chbr.2022.100213

[8] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How Do API Documentation and Static Typing Affect API Usability?. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 632–642. https://doi.org/10.1145/2568225.2568299

[9] Sarah Fakhoury, Devjeet Roy, Harry Pines, Tyler Cleveland, Cole S. Peterson, Venera Arnaoudova, Bonita Sharif, and Jonathan Maletic. 2021. gazel: Supporting Source Code Edits in Eye-Tracking Studies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 69–72. https://doi.org/10.1109/ICSE-Companion52605.2021.00038

[10] Joseph H. Goldberg and Jonathan I. Helfman. 2010. Comparing Information Graphics: A Critical Look at Eye Tracking. In *Proceedings of the 3rd BELIV'10 Workshop: BEyond Time and Errors: Novel EvaLuation Methods for Information Visualization* (Atlanta, Georgia) *(BELIV '10)*. Association for Computing Machinery, New York, NY, USA, 71–78. https://doi.org/10.1145/2110192.2110203

[11] Drew T. Guarnera, Corey A. Bryant, Ashwin Mishra, Jonathan I. Maletic, and Bonita Sharif. 2018. ITrace: Eye Tracking Infrastructure for Development Environments. In *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications* (Warsaw, Poland) *(ETRA '18)*. Association for Computing Machinery, New York, NY, USA, Article 105, 3 pages. https://doi.org/10.1145/3204493.3208343

[12] Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. 2019. When Code Completion Fails: A Case Study on Real-World Completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 960–970. https://doi.org/10.1109/ICSE.2019.00101

[13] Kenneth Holmqvist, Marcus Nyström, Richard Andersson, Richard Dewhurst, Halszka Jarodzka, and Joost Van de Weijer. 2011. *Eye tracking: A comprehensive guide to methods and measures*. OUP Oxford.

[14] Andrew Housholder, Jonathan Reaban, Aira Peregrino, Georgia Votta, and Tauheed Khan Mohd. 2022. Evaluating Accuracy of the Tobii Eye Tracker 5. In *Intelligent Human Computer Interaction*, Jong-Hoon Kim, Madhusudan Singh, Javed Khan, Uma Shanker Tiwary, Marigankar Sur, and Dhananjay Singh (Eds.). Springer International Publishing, Cham, 379–390.

[15] Saki Imai. 2022. Is GitHub copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 319–321. https://doi.org/10.1145/3510454.3522684

[16] David E. Irwin. 1992. *Visual Memory Within and Across Fixations*. Springer New York, New York, NY, 146–165. https://doi.org/10.1007/978-1-4612-2852-3_9

[17] Shaokang Jiang and Michael Coblenz. 2024. *Artifact for "An Analysis of the Costs and Benefits of Autocomplete in IDEs"*. https://doi.org/10.1145/3580435

[18] Jozsef Katona. 2022. Measuring Cognition Load Using Eye-Tracking Parameters Based on Algorithm Description Tools. *Sensors* 22, 3 (2022). https://doi.org/10.3390/s22030912

[19] Joongyum Kim, Taesik Gong, Kyungsik Han, Juho Kim, JeongGil Ko, and Sung-Ju Lee. 2020. Messaging Beyond Texts with Real-Time Image Suggestions. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services* (Oldenburg, Germany) *(MobileHCI '20)*. Association for Computing Machinery, New York, NY, USA, Article 28, 12 pages. https://doi.org/10.1145/3379503.3403553

[20] Jingxuan Li, Rui Huang, Wei Li, Kai Yao, and Weiguo Tan. 2021. Toward Less Hidden Cost of Code Completion with Acceptance and Ranking Models. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 195–205. https://doi.org/10.1109/ICSME52107.2021.00024

[21] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (<conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 52, 13 pages. https://doi.org/10.1145/3597503.3608128

[22] Xipei Liu and James Bagrow. 2019. Autocompletion interfaces make crowd workers slower, but their use promotes response diversity. *Human Computation* 6 (jun 2019), 42–55. https://doi.org/10.15346/hc.v6i1.3

[23] Roberto Minelli and Michele Lanza. 2013. Visualizing the workflow of developers. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. 1–4. https://doi.org/10.1109/VISSOFT.2013.6650531

[24] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv:2302.06590 [cs.SE]

[25] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. " It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *arXiv preprint arXiv:2304.02491* (2023).

[26] Sebastian Proksch, Sarah Nadi, Sven Amann, and Mira Mezini. 2017. Enriching in-IDE process information with fine-grained source code history. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 250–260. https://doi.org/10.1109/SANER.2017.7884626

[27] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. *SIGPLAN Not.* 49, 6 (jun 2014), 419–428. https://doi.org/10.1145/2666356.2594321

[28] Keith Rayner. 2009. The 35th Sir Frederick Bartlett Lecture: Eye movements and attention in reading, scene perception, and visual search. *Quarterly Journal of Experimental Psychology* 62, 8 (2009), 1457–1506. https://doi.org/10.1080/17470210902816461 arXiv:https://doi.org/10.1080/17470210902816461 PMID: 19449261.

[29] Vidya Setlur, Enamul Hoque, Dae Hyun Kim, and Angel X. Chang. 2020. Sneak Pique: Exploring Autocompletion as a Data Discovery Scaffold for Supporting Visual Analysis. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 966–978. https://doi.org/10.1145/3379337.3415813

[30] Bonita Sharif, Cole Peterson, Drew Guarnera, Corey Bryant, Zachary Buchanan, Vlas Zyrianov, and Jonathan Maletic. 2019. Practical Eye Tracking with iTrace. In *2019 IEEE/ACM 6th International Workshop on Eye Movements in Programming (EMIP)*. 41–42. https://doi.org/10.1109/EMIP.2019.00015

[31] Suresh Thummalapenta and Tao Xie. 2007. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 204–213. https://doi.org/10.1145/1321631.1321663

[32] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI EA '22)*. Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665

[33] Boris Velichkovsky, Markus Joos, Jens Helmert, and Sebastian Pannasch. 2005. Two visual systems and their eye movements: Evidence from static and dynamic scene perception. *Proceedings of the XXVII Conference of the Cognitive Science Society* (01 2005).

[34] WakaTime. 2023. WakaTime 2022 Programming Stats — wakatime.com. https://wakatime.com/blog/57-wakatime-2022-programming-stats. [Accessed 05-09-2023].

[35] Chaozheng Wang, Junhao Hu, Cuiyun Gao, Yu Jin, Tao Xie, Hailiang Huang, Zhenyu Lei, and Yuetang Deng. 2023. Practitioners' Expectations on Code Completion. arXiv:2301.03846 [cs.SE]

[36] Wen Zhou, Seohyun Kim, Vijayaraghavan Murali, and Gareth Ari Aye. 2022. Improving Code Autocompletion with Transfer Learning. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 161–162. https://doi.org/10.1145/3510457.3513061

[37] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing

Machinery, New York, NY, USA, 21–29. https://doi.org/10.1145/3520312.3534864

Received 2023-09-28; accepted 2024-04-16