An Empirical Study of Developer Behaviors for Validating and Repairing AI-Generated Code

Ningzhi Tang*[1], Meng Chen*[1], Zheng Ning[1], Aakash Bansal[1], Yu Huang[2], Collin McMillan[1] and Toby Jia-Jun Li†[1]

[1]*University of Notre Dame, Notre Dame, IN*
[2]*Vanderbilt University, Nashville, TN*

Abstract

Recent advances in AI-based code generation tools such as GitHub Copilot show great promise in assisting developers with programming tasks. However, there are few empirical studies that used objective measures to investigate the behavior of programmers when validating and repairing Copilot-generated codes. In this work, we conducted a user study with 9 participants using eye tracking and IDE tracking to characterize how programmers handle errors when using Copilot. We found that developers had greater cognitive effort, but were less frustrated in the editing phase of the code compared to in the understanding and navigation phases. Programmers frequently used prompts to generate code during the repair process and accepted most of the generated code, yet they scrutinized the prompt and code for validation after accepting the code. Finally, participants found several IDE features such as `Run`, `Debug`, and `GoToDeclaration` helpful for code validation.
Keywords: GitHub Copilot, code generation, programmer behavior, eye tracking, IDE tracking.

1 Introduction

Automated code generation has been a dream of programmers for decades [1], and has attracted many researchers in the communities of artificial intelligence (AI), programming language (PL), and software engineering (SE). Recently, breakthroughs in transformer-based [2] large language models (LLMs) such as GPT-3 [3] offer great possibilities for its realization. GPT-3 has shown an impressive effect on text generation and provides the basis for a new era of code generation [4]. For example, OpenAI Codex [5], a descendant of GPT-3 fine-tuned on 54 million public GitHub repositories, demonstrated promising programming performance: solving 29% of unseen Python programming tasks with only one sample per problem and 70.2% with 100 samples [6]. Recently, GitHub Copilot [7], a programming editor extension powered by OpenAI Codex, has attracted public attention and adoption [8].

The emergence of AI-generated code means a new task for programmers: integrating AI-generated code into existing codebases. The usual situation is that a programmer asks the AI tool to generate a block of code (perhaps 10–20 lines); then the programmer must validate that code to ensure that it does not contain errors and will integrate correctly with the context of the code. At present, this process of validating AI-generated code is not well understood. AI tools may generate different types of errors than fellow humans, and unlike humans, the AI is not able to articulate the rationale behind its decisions, so the debugging strategy people follow may be different from when debugging human-written code. It is important to know how programmers understand and repair errors in AI-generated code to help guide the development of the underlying AI models and their interfaces. Various studies have evaluated the performance of AI-based code generators against benchmarks [6], [9] and specific scenarios [10], [11]. Other studies have provided early evidence about the behavior patterns of programmers when using Copilot [12], [13]. While they are intriguing early evidence, more light needs to be shed on the process of human-AI collaboration and how it differs from past work processes.

In this paper, we present an empirical study of programmer behaviors during the validation of AI-generated code using eye tracking and IDE tracking. The purpose of our study was to investigate the programmers' cognitive workload, validating strategies, and collaboration with GitHub Copilot [7] in the processes. We conducted studies with 9 participants, in which they needed to validate and integrate Copilot-generated codes into three different software projects. In the study, participants validated and repaired errors with the help of Copilot. We collected the eye-tracking data and IDE behavior data using a plugin for IntelliJ IDEA [14] that we developed. The study investigated the following research questions:

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

*These authors contributed equally to this work.
†Email: toby.j.li@nd.edu

- **RQ1:** How much cognitive workload do programmers have during the error discovery and repair process?
- **RQ2:** What strategies do programmers use to handle errors in code generated by Copilot?
- **RQ3:** What is the role of Copilot in the process of handling errors in Copilot-generated codes?

2 Related Work

2.1 AI-Enabled Code Generation

Recently, advances in deep learning, especially LLMs such as GPT-3 [3], have fueled the development of automated code generation. For example, AlphaCode [15], which focused on competitive programming problems that require deeper reasoning, achieved on average a ranking of the top 54.3% in popular competitions. GitHub Copilot [7], which is based on Codex [5], a fine-tuned version of GPT-3, has shown great performance in automatic code generation. It has greatly improved developers' productivity and programming speed [16], which motivated us to research the collaboration of programmers with it.

Previous studies have evaluated their performance on benchmark datasets with various problem-solving scenarios. For example, Chen et al. [6] used HumanEval to measure the functional correctness of Codex synthesized programs from docstrings. Nguyen et al. [9] used LeetCode questions to create queries for Copilot and evaluated the correctness of the corresponding solutions. Pearce [11] evaluated the performance of LLM-based code tools to help repair human-generated bugs. Finnie-Ansley et al. [10] measured the ability of Codex in introductory programming education.

However, these studies focused mainly on generation performance. It is still unknown how users interact with the tools. Current empirical studies are only based on direct observations of Copilot usage, lack concrete behavior data for more objective quantitative analysis, and did not focus on understanding the validation process. Our study bridges this gap in the literature by incorporating eye tracking and IDE behavior tracking to characterize programmers' code validation process with Copilot.

2.2 Eye Tracking for Software Engineering

Eye tracking is a process of capturing human visual attention by measuring eye gaze data [17]. It has been widely used in HCI to understand and model human cognitive processes [18], [19]. Eye-related features, such as blink rate, pupillary response, and fixation information, can serve as indicators of mental effort in various activities, including SE tasks [20], [21].

Eye tracking technology has been used in software engineering research to study human behavior during programming since Crosby et al. [22] first studied the eye movements of programmers in 1990. A survey by Sharafi et al. [17] chronicles up to 2015 and is expanded by Obaidellah et al. [23]. Eye tracking data is widely recognized to represent the attention given to different parts of the source code by humans. Previous work has shown that a longer fixation time of an area indicates rich information and greater complexity of the element [24], [25]. An exemplary body of work is led by Sharifi et al., with advances in both knowledge of how programmers read code and practical uses for this knowledge [26], [27]. Eye tracking in SE is often used to complement and enhance behavior tracking through an integrated development environment (IDE) [24]. The idea is to monitor both what developers look at and what they do (e.g., mouse clicks, file navigation, and attempts to compile) [28]–[31].

2.3 Human-AI Collaboration in Software Development

There were some previous empirical studies on the usage of GitHub Copilot. Vaithilingam et al. [12] conducted a user study to understand how programmers use and perceive Copilot. Barke et al. [13] presented a grounded theory analysis of how programmings interacted with Copilot through observations on their solving of programming tasks in four languages. However, these works were summarized mainly through qualitative research methods such as manual observation and interviews. The lack of quantitative analysis and objective methods to measure programmer behaviors (e.g., eye tracking and IDE behavior logging) present a gap in the literature.

Beyond Copilot, in relation to other LLM-based code generation tools, Jiang et al. [32] conducted a user study in which participants applied GenLine to programming tasks. GenLine is different from Copilot because they explicitly invoke a command in a text editor. Weisz et al. [33] interviewed 11

IBM software engineers on a programming language translation tool about their tolerance degree of imperfections and ways to aid in debugging errors.

2.4 Debugging Strategies of Software Developers

Another closely related area is the debugging strategies of software developers. The study of how programmers comprehend, write, and debug codes is a long-standing topic in SE research [34], [35]. Human debugging strategies shed light on the design of debugging tools. Moritz Beller et al. [36] found the limited usage of IDE features and proposed to improve the Eclipse debugger based on observation. Lawrance et al. [37] use information foraging theory to construct a model that predicts programmer navigation choices during debugging, suggesting that a debugging tool should support scent following through the information patch by providing proximal cues. Debugging strategies also contribute to the validation of debugging tools [38]. Many previous works have evaluated the validity of debugging tools by testing whether they fit in the debugging process [39]–[41].

However, our domain is fundamentally different because the study of debugging strategies focuses on debugging codes either generated by programmers themselves or their human colleagues, which are quite different from understanding and validating codes generated by AI models. AI-generated codes are also different from those generated by humans. The results of a previous empirical study showed that Copilot-generated code is not as compact as human-created code [9]. A study also found that the Copilot-generated code is of lower quality than the human-created code [42]. We suspect that the error handling strategies used in human-AI collaboration in software development might share some similar characteristics to those in error handling for other human-AI interaction contexts, e.g., conversational agents [43] qualitative coding [44], and natural language data queries [45].

3 Study Design

We designed a lab study to understand how users validate the code generated by GitHub Copilot. In our study, we used three programming tasks (details in Section 3.1). For each task, we prepare a codebase with several declared but not implemented classes/methods. For each of them, we came up with prompts that described its functionality. We deliberately chose prompts for which Copilot will generate codes with representative errors of different types, as described in Section 3.1).

The codebase, along with the prompts and Copilot-generated code, was provided to each participant to validate and repair the errors (so each user started from exactly the same codebase). The detailed protocol of our study is described in Section 3.4.

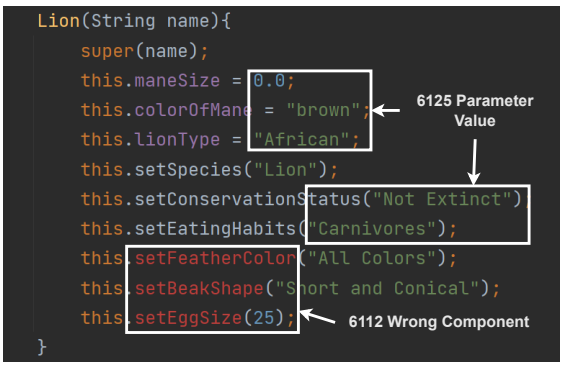

Figure 1. A bug example from subtask 3.2: overload constructors.

3.1 Programming Tasks

We used three Java programming tasks in different software engineering scenarios: algorithm design, graphical user interface (GUI) design, and object-oriented programming (OOP).

- **Task 1.** *Kakaroma (Algorithm Design)*: A LeetCode-like task that utilizes dynamic programming to find a path in a square array that has the minimum sum of numbers. The entire program was written by Copilot from scratch. This task was adapted from an assignment from a computer science undergraduate course [46].

- **Task 2.** *Calculator (GUI):* A calculator app written with the Java GUI programming API, consisting of a front-end that includes a text interface and operating buttons, as well as back-end logic. The GUI layout code and the listeners for front-end buttons were generated by Copilot.
- **Task 3.** *ZooSystem (OOP):* A zoo management system that has various animal classes with inheritance relationships (e.g., animal-mammal-lion). Several management functions, such as add, delete, search, and display animals, are generated by Copilot. This task was adapted from an assignment from an undergraduate computer science course [47].

We categorized these errors based on the taxonomy in *Software Testing Technique* [48]. The error statistics for three tasks are in Table 1.

Table 1. Taxonomy of the bugs existing in programming tasks based on [48].

Task	No.	Subtask	Bug index [48]	Bug category [48]
Kakaroma	1.1	create table	231x	Missing Case
	1.2	reconstruct path	3226.4	String Manipulation-Insertion
			3126	Illogic Predicates
			231x	Missing Case
Calculator	2.1	GUI layout	6125	Parameter Value
	2.2	set listener	614x	Initialization State
ZooSystem	3.1	set attribute value	6125	Parameter Value
	3.2	overload constructors	6112	Wrong Component
			6125	Parameter Value
	3.3	dynamic array	4164	Should be Dynamic Resource
	3.4	system input method	6112	Wrong Component
	3.5	attribute name	413x	Initial, Default Values

For example, Fig. 1 shows bugs in the code generated by Copilot in subtask 3.2, with the prompt "create a constructor for the Lion class that only takes name as input". The code has both bug type 6125 "Parameter Value" (e.g., the conservation status should be "Vulnerable" by default), and bug type 6112 "Wrong Component" (the lions do not have features, it should be fur color extending from the superclass Mammal).

3.2 Study Settings

We conducted the study in person in a usability lab on a computer with a 27-inch monitor and a Tobii Pro Fusion [49] eye tracker with the sampling frequency set at 60 Hz. The user interacted with the code and Copilot through IntelliJ IDEA 2022.1.4 running the Copilot plugin and our data collector plugin (Section 3.5). We set the font size of the IDE at 20 points to mitigate the impact of the drifting effect of the eye tracker on the location of the code tokens corresponding to the eye gazes. To mitigate the influence of light intensity on eye tracking, all study sessions were held in the same room with all doors and windows closed and the same ceiling light on.

3.3 Participants

We recruited 9 participants (5 female, 4 male) from the local university community. 2 participants were undergraduate students, and 7 were graduate students. 1 of them had no programming experience in Java before, 4 of them had just taken an introductory course on Java programming, and 4 had 1∼3 years of programming experience. Only 2 participants had used Copilot prior to our study. Participants received a $30 Amazon Gift Card as compensation for their time.

3.4 Protocol

The study took approximately two hours per participant. After signing the consent form and filling out the pre-study questionnaire that collected the participant's demographic data, we told the participant about the overall objective and process of the study, followed by a 10-minute tutorial on the usage of

IntelliJ IDEA and Copilot. We also gave a short instruction on interacting with the eye tracker (e.g., refraining from major head movements).

For each task, the participant read the instruction describing the background and required tasks first and calibrated the eye tracker. The participant then had 20 minutes to perform the task. To narrow down the debugging strategy for specific errors, participants were asked to self-report when they thought they had completed a subtask. Participants were allowed to ask clarification questions to the experimenter, but were not allowed to browse the Internet. After completing each task, participants were asked to complete a NASA TLX form [50] to self-report their cognitive workload. In the end, participants shared their experience performing the tasks in a 15-minute semi-structured interview. The interview included questions regarding participants' bug validating and repairing strategies, usage of Copilot functions and IDE features, as well as their different feelings about debugging Copilot-generated code and code written by humans. We used Grounded Theory to analyze our qualitative data. We open-coded valuable insights from the interview transcripts and experimenter observations and conducted axial coding to organize them into our findings.

3.5 Data Collection Setup

To support the analysis of programmer behaviors, we developed a plugin for IntelliJ IDEA for IDE tracking and eye tracking. We also made screen recordings with timestamps for study sessions for subsequent analysis. The detail of the data collected is summarized in the following sections.

3.5.1 IDE Tracking

Our plugin collects the following information via IDE tracking. All behaviors are tracked with their location (path, line, column) and timestamp for further analysis and calibration. They are organized in XML format and an example is shown in Fig. 2.

IDE Features

```
<action id="ReformatCode"
path="/src/Reptile.java"
timestamp="1662928377436"/>

<action id="copilot.applyInlays"
path="/src/Reptile.java"
timestamp="1662928415454"/>

<action column="23" line="42"
path="/src/Reptile.java"
id="GotoDeclaration"
timestamp="1662928462655"/>

<action path="/src/Reptile.java"
id="SaveAll"
timestamp="1662928462698"/>

<action id="EditorPaste"
path="/src/Lion.java"
timestamp="1662928472075"/>
```

Keyboard Actions

```
<action id="EditorBackSpace"
path="/src/ZooSystem.java"
timestamp="1662928386767"/>

<typing column="4" length="13"
line="8" path="/src/Reptile.java"
string="private int E"
timestamp="1662928450732"/>

<action id="EditorDown"
path="/src/ZooSystem.java"
timestamp="1662928678575"/>

<typing column="24" length="4"
line="32" path="cmd" string="Fido"
timestamp="1662928789679"/>

<scroll offset="693" type="vertical"
path="/src/ZooSystem.java"
timestamp="1662928825827"/>
```

File Logging

```
<log id="fileLog" info="fileOpened"
src_path="/src/ZooSystem.java"
timestamp="1662928820593"/>

<log id="fileLog" info="fileChanged"
src_path="/src/Lion.java"
timestamp="1662928828525"/>

<log id="fileLog" info="fileChanged"
src_path="/src/Lion.java"
timestamp="1662928829436"/>

<log id="fileLog" info="fileClosed"
src_path="/src/Lion.java"
timestamp="1662928831515"/>

<log id="outputLog"
timestamp="1662928831632"/>
```

Figure 2. An example of IDE tracking data format. The left part is IDE features, the middle part is keyboard typing, and the right part is file logging.

IDE Features Our plugin recorded any IDE features used during programming. These include (1) *IDE support edit*: `ReformatCode`, `CommentByLineComment`, `Copy`, `Cut`, `Paste`; (2) *run and debug action*: `RunClass`,`Stop`, `Debug`, `ToggleLineBreakPoint`; (3) *file action*: `Open`, `SaveAll`, `NewFile`; (4) *find/replace and code inspection*: `Find`, `GotoDeclaration`, `CompareTwoFiles`. The IDE features usage log also included the actions of Copilot usage, such as `copilot.applyInlays` representing accepting the code generated by Copilot.

Keyboard Actions Our plugin collected all code edits using the keyboard. It could also detect keyboard actions with special functions: (1) editing function, such as `Enter`, `Tab`, `Backspace`, `Delete`; (2) navigation function like `Left`, `Right`, `Up`, and `Down`; (3) selection function, such as `SelectWord`, `SelectLine`. In addition, mouse scrolling behaviors (vertical/horizontal) were recorded.

File Logging In order to recover any code file at any timestamp during programming tasks, we

designed a file logging mechanism that logs the whole content of the code file when (1) the programmer opens/closes a code file or (2) a code file is edited during the past 1 second. Our plugin also records the console output when the programmer executes the code.

3.5.2 Eye Tracking

Our plugin collects raw gaze data and maps them to semantic source code entities, such as tokens and nodes in abstract syntax trees (ASTs). The sampling frequency of the eye tracker is 60 Hz. A sample of gaze data and the calculated semantic source code entities is shown in Fig. 3.

```
<gaze column="12" gaze_validity="1.0" left_pupil_diameter="3.277679443359375" line="26" path="/src/ZooSystem.java"
      pupil_validity="1.0" right_pupil_diameter="3.2931365966796875" x="545" y="306" log_timestamp="1669258001910"
      end_timestamp="1669258026693" count="18" start_timestamp="1669258026407" duration="286">
    <psi ast_type="IDENTIFIER" token="printSummaryView">
        <level end="26:24" start="26:8" tag="PsiIdentifier:printSummaryView"/>
        <level end="26:24" start="26:8" tag="PsiReferenceExpression:printSummaryView"/>
        <level end="26:31" start="26:8" tag="PsiMethodCallExpression:printSummaryView(alist)"/>
        <level end="26:32" start="26:8" tag="PsiExpressionStatement"/>
        <level end="32:5" start="5:4" tag="PsiMethod:setupAnimals"/>
        <level end="499:27" start="3:0" tag="PsiClass:ZooSystem"/>
    </psi>
</gaze>
```

Figure 3. An example of raw gaze data collected from the eye tracker, which contains the computed location, token, as well as its AST hierarchy (PSI).

Raw Gaze Data The raw data received from the eye tracker include: (1) (x, y) relative coordinates on the screen multiplied by the width and height of the screen, as well as (2) left/right pupil diameters and their corresponding data validities. In the Tobii Pro SDK, each data relating to the eye is provided with its own validity code. Invalid data were excluded from our analysis based on the recommendation of Tobii Pro [51]. Each gaze is recorded with a timestamp.

Location, Token & AST To extract semantic information from gazes, such as their corresponding code tokens, we first calculated the relative location in the code editor and then mapped it to a specific location (line, column) of the code file and its corresponding token. Finally, we searched for the token in all the parent nodes of the AST with the Program Structure Interface (PSI) [52] in IntelliJ IDEA.

Fixation & Saccade Fixation is eye stabilization at one location for a period of time. We extracted gazes on the same token with durations longer than 200 ms as fixations. We also take the transitions between two fixations within 50 ms as saccades [17].

4 Study Result

This section describes the quantitative and qualitative results of our study based on our three RQs. We summarize the descriptive statistics on the overall performance of the users in Table 2.

Table 2. Overall performance of users.

Task	No.	Subtask	Avg. time spent (min.)	Success rate
Kakaroma	1.1	create table	8.13	0.88
	1.2	reconstruct path	11.75	0.25
Calculator	2.1	GUI layout	20.38	0.5
	2.2	set listener	5	1
ZooSystem	3.1	set attribute value	3.13	0.88
	3.2	overload constructors	6.38	1
	3.3	dynamic array	9	0.63
	3.4	system input method	2.6	1
	3.5	attribute name	3.66	0.67

4.1 Programming Activities Classification

To study the programming activities of the participants, we classified their behaviors using the methods described by Minelli et al. [53]. We measured the time developers spent understanding, navigating, and editing source code based on our collected data.

A researcher quantitatively annotated the programmer's behaviors as ***understanding***, ***navigation***, and ***editing*** events based on tracked IDE behaviors. To illustrate an example of programming activities classification on the timeline, we show a glance at a 400-second session in Fig. 4.

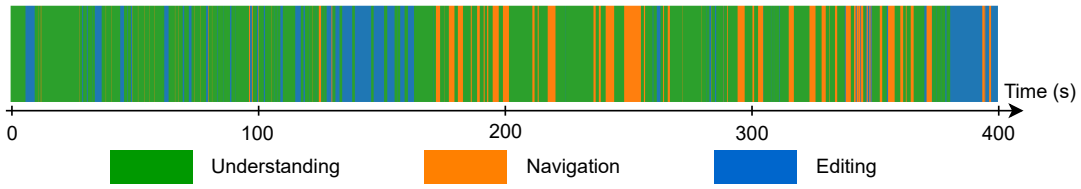

Figure 4. A glance of programming activities classification. This session lasted about 400 seconds.

Navigation The code navigation behaviors (orange part in Figure 4) refer to the search and locating of the code. There are two types of navigation behaviors: (1) *Across-file navigation:* includes actions such as opening, closing, or changing the selection of code files. (2) *Within-file navigation:* includes vertical/horizontal scrolling within one file, the use of relevant IDE features (e.g., code inspection and find & replace as defined in Table 5), and keyboard actions (e.g., arrow keys).

Editing We tracked code editing behaviors (blue part in Figure 4) based on recorded keystrokes, the use of relevant IDE features (editor selection, IDE support editing as defined in Table 5), and the use of Copilot to generate or edit codes.

Understanding The understanding behaviors (green part in Figure 4) represent the time that users spend looking at the code without navigating or editing behaviors.

On average, participants spent 85.7% of their time in the understanding phase, 10.3% in the navigation phase, and 4.0% in the editing phase.

4.2 Workload Comparison (RQ1)

To measure the workload of the programmer when validating the code generated by Copilot, we used both eye-tracking and self-reported questionnaires and compared their results. Our findings suggest that programmers have a higher visual workload in the editing phase compared to the understanding and navigation phases, and the visual workload measured by pupil diameters is negatively correlated with their self-reported frustration levels.

In eye tracking, pupil diameters were commonly used to measure the visual workload of participants and were known to be correlated with cognitive workload [17]. Larger pupil diameters indicate greater visual effort [54]. Our data recorded the pupil sizes of the participants when performing our tasks, as shown in Fig. 5. We calculated the correlation between the pupil diameters of the left and right eyes for all users. The data suggest that they are highly correlated (Pearson correlation 0.958). Therefore, we use the data on the average pupil diameter of the left and right eyes for further analysis [55]. Following best practice, when processing the pupil diameter data, we use 2 mm as the threshold to filter out the noise caused by blinking (0.064% of the collected data) [56].

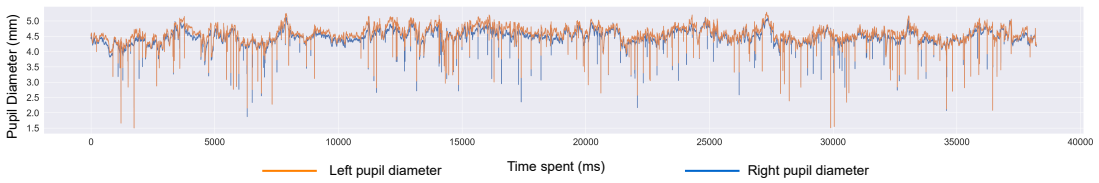

Figure 5. A glance of left and right pupil diameters over time.

We aggregated the pupil diameters of user gazes on three programming activities and present the results in box plots for different programming tasks (Fig. 6). Participants' workload in the editing

Table 3. Results of self-reported workload in NASA-TLX questionnaires. *Top:* Percentage of self-reported high (or very high) understanding, navigation, and editing effort. *Bottom:* average ratings of all participants over three tasks. Regarding performance, a lower score means perfect and a higher score means failure.

Category	Metric	Kakamora	Calculator	ZooSystem
Activities	High understanding effort (%)	55.6	11.1	77.8
	High navigation effort (%)	66.7	33.3	44.4
	High editing effort (%)	55.6	44.4	11.1
NASA TLX	Mental Demand	7.1	6.9	5.4
	Physical Demand	3.3	3.7	3.2
	Temporal Demand	5.9	5.9	4.6
	Performance	5.4	6	5
	Effort	6.4	7	5.7
	Frustration	4.8	5.8	4.4

phase, measured by pupil diameters, is greater than that in the understanding ($4.213 > 4.073$, $p < 0.001$) and navigation phases ($4.213 > 4.07$, $p < 0.001$). This difference is consistent in all three tasks (Kakamora: $4.312 > 4.22/4.208$, $p < 0.05$; Calculator: $4.161 > 3.937/3.954$, $p < 0.001$; ZooSystem: $4.16 > 4.046/4.035$, $p < 0.01$). All p-values are computed by the Student's t-test. There is no significant difference in the pupil diameter of users between the navigation and understanding phases.

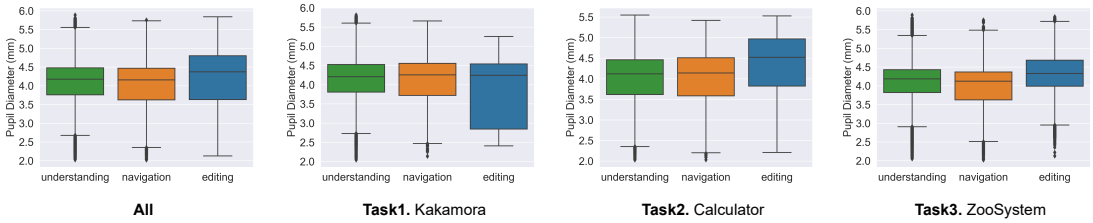

Figure 6. Participants' pupil diameters in understanding, navigation, and editing phases.

We asked all users to self-report their workload after finishing each task using a NASA TLX questionnaire and report their effort in three programming activities. The results are summarized in Table 3. We computed the Pearson correlation between pupil diameters in three programming activities with self-reported data. The result shows that frustration in NASA TLX is negatively correlated with visual efforts measured by pupil diameters in all three activities (-0.455/-0.466/-0.462, $p < 0.05$). The other pairs of relationships were not statistically significant.

4.3 Validating Strategy (RQ2)

In this section, we explore the debugging strategies of programmers by analyzing token fixes, as well as the use of IDE features and Copilot. We found that programmers had a high concentration on the prompt used in Copilot and frequently switched between the comment prompts and code. Programmers often used the run, debug, and `GotoDeclaration` features, which were also often combined with a fixation on identifiers. Furthermore, many interactions with Copilot were continuous, while part of them are often related to deleting the code that was just generated.

4.3.1 Fixations in Programming Tasks

The average fixation count (FC) of a participant in a task is 525.4 ms, with an average fixation duration (AFD) of 467 ms. The total fixation time per task is 244.4 seconds, which reflects the visual attention of the participants. On average, a participant triggered 84.1 saccades per task with an average saccade length (ASL) of 96.67 pixels, which reflects the search effort of programmers and contains limited visual perception. Since fixation is related to information processing and is important for eye tracking analysis, we analyze their patterns with respect to abstract syntax tree (AST) structures in code to study the programmers' visual strategy of debugging. We aggregated the fixation time on different

token types, and counted the bigram of all users' sequences of fixations by token types. An *n*-gram is a contiguous sequence of *n* items from a given sequence. The top-5 results are summarized in Table 4.

Among the different types of tokens, the identifiers and comments attracted the most fixations. All comments here are prompts either given by us or written by participants to generate code (instead of regular documentation) in our task setting. This reflects that (1) programmers had a high concentration on understanding the prompts themselves (the 2nd most frequent bigram), and (2) programmers also spent much of their time switching back and forth between the comment prompt and the generated code (the 3rd and 4th most frequent bigrams).

4.3.2 Usage of IDE Features

We summarized the usage of the IDE features for three programming tasks in Table 5. We combined features with similar functionalities together (e.g., Run, RunClass, and Rerun) and filtered out the seldom used features, i.e., the total number of usage ≤ 5. The result shows that participants often validated the codes generated by Copilot in two ways: running the class and using the debugger. But the distribution of their usage differed in different tasks. In the interviews, the participants reported that the use of the debugger was more useful in our algorithm design task, while tracking the bugs according to the error message from running the class was more helpful to validate the GUI task. P7 describe their strategy as *"eyeball code first and then run it to see error messages"*.

We also examined the adjacent fixations just before or after the use of IDE features. The top-4 combinations were (GotoDeclaration, *identifier, GotoDeclaration), (*identifier, Resume), (*identifier, StepOver), and (RunClass, *identifier). The element with * indicates the AST type of the token that corresponds to the fixation.

4.3.3 Usage of Copilot

There were 302 `copilot.applyInlays` usages in total, which was more than any of the other IDE features. We counted the trigrams of the mixture of Copilot usage and keyboard typing. The result shows that almost all of the top-10 frequent trigrams are combinations of `copilot.applyInlays`, `EditorEnter`, and `EditorBackspace`. In addition, 61.6% of these trigrams were continuous code "generating–accepting" sequences, and the other 38.4% contain backspaces that deleted the generated code. This reflects that the interaction between the programmer and Copilot is usually continuous, with some cases where the generated code is quickly deleted by the developer.

4.4 Programmer-Copilot Collaboration (RQ3)

In this section, we explored the features of programmer-Copilot collaboration via the analysis of code generation sources, code acceptance rate, and feedback from semi-structured interviews. We found that users started to rely more on Copilot once they get familiar with Copilot. We also identified two ways of collaboration between programmers and Copilot.

4.4.1 Collaboration Model

Participants found Copilot helpful during the debugging process, and 74% of the participants agreed that the Copilot-generated code is reasonable. Copilot reduced the low efficiency and inconvenience of switching windows and consulting the documentation. P6 said, *"[Copilot] avoid the distraction of switching windows"*. Participants generally thought that the Copilot code looks similar to human-written code in terms of style and quality. P1 said that *"Code in the algorithm task is similar to human-written one. Yet, codes generated in the Calculator and Zoo System tasks were not like those written by humans because they involved some undefined features"*.

Participants reported two ways of collaboration. First, most of the participants found that generating code by comments was a useful way to start from scratch for a block of code. However, programmers need to thoroughly understand the task of writing an accurate natural language prompt. P4 said *"It takes some effort for me to think of the prompt"* and P7 suggested that *"using comment that includes the context can generate code of higher quality"*. Second, participants used Copilot to repair minor bugs. Especially those who are not familiar with Java found that Copilot was useful for

fixing errors in syntax and data types. Participants preferred one-line-long generation over a large chunk of code as they found the latter *"cumbersome"* as it still required significant effort to understand the meanings of the generated code. P4 said *"I found it hard to accept Copilot generated code at first because if it is intuitive, I can write on my own; if it is not, I have to read it"*.

4.4.2 Reliance on Copilot

When Copilot was available, the programmers in our study tended to use it to validate and repair the code. The code created during the study came from three sources: (1) Copilot, mainly via `copilot.applyInlays` action. (2) IDE features in Table 5, such as copying and pasting. (3) Typing on keyboards. We summarize the average number of instances of each type, the average number of characters in the generated code, and the percentage of sources that generated the most characters among all participants in Table 6. The result shows that Copilot generated the most number of characters in our tasks. The result is aligned with our interview finding that Copilot often served as a replacement for copy/paste actions. P8 said *"I changed my approach. First, I saw if Copilot could give me some useful suggestions, and then I validated them"*.

Participants kept most of the model-generated code in the final code. We compared the model-generated code and the final code texts in similarity (by Gestalt pattern matching) and distance (by Levenshtein distance). Gestalt pattern matching measures the longest contiguous matching subsequence that contains no "junk" elements (e.g., blank line, white space). Levenshtein distance measures the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one text to the other [57]. Our results show that the average Gestalt similarity between the model-generated code and the final code is 0.77 (0.82/0.65/0.85 for each task). The average Levenshtein distances between the model-generated code and the final code produced by the study participants are 231.67, 703, and 425 for the three tasks (Kakaroma, Calculator, ZooSystem) respectively. The initial lengths of code generated by Copilot are 1194, 2507, and 3551 for the three tasks respectively.

Three participants mentioned that Copilot changed their roles in the programming process. P7 said that *"coder's responsibility changed to guide Copilot writing something logically and coherently"*. Compared to writing code manually, collaborating with Copilot can help mitigate the bias in debugging programmers' own codes, as they may fall into the same trap all the time. Participants said that they are more critical when validating and repairing Copilot-generated code. P5 said, *"debugging with Copilot was easier since it had a completely different thinking angle"*.

5 Limitation and Future Work

As a preliminary empirical study, this work has presented several findings on how developers handle errors when working with generative code models and provided implications for the design of future AI-enabled programming support tools. However, there are still some limitations in the study and many opportunities for future work to further validate and expand our findings.

First, we plan to conduct a *controlled* study to compare the user behaviors we observed in this study with those when users perform the same tasks without Copilot. This will allow us to measure the impact of using Copilot on user cognitive effort and directly compare user cognitive processes and strategies when they use Copilot with those when they write code manually for the same tasks.

Second, the generalizability of our findings is threatened by the limited representation of participant groups, task types, and bug types. To address this, we plan to conduct a more robust, more comprehensive, and larger-scale study in the future. For example, despite that we chose three representative programming tasks that cover bugs from the algorithmic level to the component invocation level, we could include more categories of bugs and categories of programming tasks. The limited time and lab setting constrained the number of bugs and categories of tasks that we can include in our experiment. In the future, we can conduct a longitudinal field study in which programmers participate for months and build a large program at their own pace. Furthermore, many of our participants had no prior experience using Copilot. Familiarity with Copilot may affect participants' behaviors.

Lastly, there are also limitations to the nature of eye tracking. Eye trackers can capture what users are looking at, but cannot reveal what and how the users think. To mitigate this risk, we integrate

self-report and semi-structured interviews to enhance our findings.

Acknowledgement

This research was supported in part by an AnalytiXIN Faculty Fellowship, an NVIDIA Academic Hardware Grant, a Google Cloud Research Credit Award, a Google Research Scholar Award, and NSF grants CCF-2211428 and CCF-2100035. Any opinions, findings, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the sponsors. We thank Robert Wallace for his assistance in setting up the eye tracker.

References

[1] R. J. Waldinger and R. C. Lee, "Prow: A step toward automatic program writing," in *Proceedings of the 1st international joint conference on Artificial intelligence*, 1969, pp. 241–252.

[2] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[3] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[4] OpenAI, *Gpt-3 powers the next generation of apps*, May 2022. [Online]. Available: https://openai.com/blog/gpt-3-apps/.

[5] W. Zaremba, *Openai codex*, Nov. 2021. [Online]. Available: https://openai.com/blog/openai-codex/.

[6] M. Chen, J. Tworek, H. Jun, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] *Github copilot · your ai pair programmer*. [Online]. Available: https://github.com/features/copilot.

[8] C. Metz, *A.i. can now write its own computer code. that's good news for humans.* Sep. 2021. [Online]. Available: https://www.nytimes.com/2021/09/09/technology/codex-artificial-intelligence-coding.html.

[9] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[10] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather, "The robots are coming: Exploring the implications of openai codex on introductory programming," in *Australasian Computing Education Conference*, 2022, pp. 10–19.

[11] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Can openai codex and other large language models help us fix security bugs?" *arXiv preprint arXiv:2112.02125*, 2021.

[12] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, 2022, pp. 1–7.

[13] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *arXiv preprint arXiv:2206.15000*, 2022.

[14] *Intellij idea – the leading java and kotlin ide*. [Online]. Available: https://www.jetbrains.com/idea/.

[15] Y. Li, D. Choi, J. Chung, *et al.*, "Competition-level code generation with alphacode," *arXiv preprint arXiv:2203.07814*, 2022.

[16] E. Kalliamvakou, *Research: Quantifying github copilot's impact on developer productivity and happiness*, Sep. 2022. [Online]. Available: https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/.

[17] Z. Sharafi, T. Shaffer, B. Sharif, and Y.-G. Guéhéneuc, "Eye-tracking metrics in software engineering," in *2015 Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2015, pp. 96–103.

[18] S. Chen, J. Epps, N. Ruiz, and F. Chen, "Eye activity as a measure of human mental effort in hci," in *Proceedings of the 16th international conference on Intelligent user interfaces*, 2011, pp. 315–318.

[19] B. Pfleging, D. K. Fekety, A. Schmidt, and A. L. Kun, "A model relating pupil diameter to mental workload and lighting conditions," in *Proceedings of the 2016 CHI conference on human factors in computing systems*, 2016, pp. 5776–5788.

[20] J. G. May, R. S. Kennedy, M. C. Williams, W. P. Dunlap, and J. R. Brannan, "Eye movement indices of mental workload," *Acta psychologica*, vol. 75, no. 1, pp. 75–89, 1990.

[21] J. Zagermann, U. Pfeil, and H. Reiterer, "Measuring cognitive load using eye tracking technology in visual computing," in *Proceedings of the sixth workshop on beyond time and errors on novel evaluation methods for visualization*, 2016, pp. 78–85.

[22] M. E. Crosby and J. Stelovsky, "How do we read algorithms? a case study," *Computer*, vol. 23, no. 1, pp. 25–35, 1990.

[23] U. Obaidellah, M. Al Haek, and P. C.-H. Cheng, "A survey on the usage of eye-tracking in computer programming," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–58, 2018.

[24] N. Ali, Z. Sharafi, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study on the importance of source code entities for requirements traceability," *Empirical software engineering*, vol. 20, no. 2, pp. 442–478, 2015.

[25] T. Busjahn, R. Bednarik, and C. Schulte, "What influences dwell time during source code reading? analysis of element type and frequency as factors," in *Proceedings of the Symposium on Eye Tracking Research and Applications*, 2014, pp. 335–338.

[26] B. Sharif, M. Falcone, and J. I. Maletic, "An eye-tracking study on the role of scan time in finding source code defects," in *Proceedings of the Symposium on Eye Tracking Research and Applications*, 2012, pp. 381–384.

[27] B. Sharif, J. Meinken, T. Shaffer, and H. Kagdi, "Eye movements in software traceability link recovery," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1063–1102, 2017.

[28] Z. Sharafi, I. Bertram, M. Flanagan, and W. Weimer, "Eyes on code: A study on developers code navigation strategies," *IEEE Transactions on Software Engineering*, 2020.

[29] S. Jiang, C. McMillan, and R. Santelices, "Do programmers do change impact analysis in debugging?" *Empirical Software Engineering*, vol. 22, no. 2, pp. 631–669, 2017.

[30] P. Hejmady and N. H. Narayanan, "Visual attention patterns during program debugging with an ide," in *proceedings of the symposium on eye tracking research and applications*, 2012, pp. 197–200.

[31] S. Maan, "Representational learning approach for predicting developer expertise using eye movements," 2020.

[32] E. Jiang, E. Toh, A. Molina, *et al.*, "Discovering the syntax and strategies of natural language programming with generative language models," in *CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–19.

[33] J. D. Weisz, M. Muller, S. Houde, *et al.*, "Perfection not required? human-ai partnerships in code translation," in *26th International Conference on Intelligent User Interfaces*, 2021, pp. 402–412.

[34] R. Brooks, "Towards a theory of the cognitive processes in computer programming," *International Journal of Human-Computer Studies*, vol. 51, no. 2, pp. 197–211, 1999.

[35] A. von Mayrhauser and A. M. Vans, "Program understanding needs during corrective maintenance of large scale software," in *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, IEEE, 1997, pp. 630–637.

[36] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 572–583.

[37] 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, 2010.

[38] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 117–128.

[39] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: A human study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 64–74.

[40] W. Dou, S.-C. Cheung, and J. Wei, "Is spreadsheet ambiguity harmful? detecting and repairing spreadsheet smells due to ambiguous computation," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 848–858.

[41] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 802–811.

[42] S. Imai, "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*, 2022, pp. 319–321.

[43] T. J.-J. Li, J. Chen, H. Xia, T. M. Mitchell, and B. A. Myers, "Multi-Modal Repairs of Conversational Breakdowns in Task-Oriented Dialogs," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST 2020, ACM, 2020. DOI: 10.1145/3379337.3415820.

[44] S. A. Gebreegziabher, Z. Zhang, X. Tang, Y. Meng, E. Glassman, and T. J.-J. Li, "Patat: Human-ai collaborative qualitative coding with explainable interactive rule synthesis," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. CHI '23, ACM, 2023.

[45] Z. Ning, Z. Zhang, T. Sun, Y. Tian, T. Zhang, and T. J.-J. Li, "An empirical study of model errors and user error discovery and repair strategies in natural language database queries," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*, ser. IUI '23, 2023.

[46] *Challenge 12: Kakamora*. [Online]. Available: https://www3.nd.edu/~pbui/teaching/cse.30872.fa22/challenge12.html.

[47] *Programming paradigms summer 2022*. [Online]. Available: https://www3.nd.edu/~skumar5/teaching/2022-summer-pp.html.

[48] B. Beizer, *Software testing techniques*. Dreamtech Press, 2003.

[49] *Reach further with your research: Choose tobii pro fusion*. [Online]. Available: https://www.tobii.com/products/eye-trackers/screen-based/tobii-pro-fusion.

[50] S. G. Hart, "Nasa task load index (tlx)," 1986.

[51] *Validity codes*. [Online]. Available: https://developer.tobiipro.com/commonconcepts/validitycodes.html.

[52] *Program structure interface (psi): Intellij platform plugin sdk*. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/psi.html.

[53] R. Minelli, A. Mocci, M. Lanza, and T. Kobayashi, "Quantifying program comprehension with interaction data," in *2014 14th International Conference on Quality Software*, IEEE, 2014, pp. 276–285.

[54] A. Poole and L. J. Ball, "Eye tracking in hci and usability research," in *Encyclopedia of human computer interaction*, IGI global, 2006, pp. 211–219.

[55] K. Krejtz, A. T. Duchowski, A. Niedzielska, C. Biele, and I. Krejtz, "Eye tracking cognitive load using pupil diameter and microsaccades with fixed gaze," *PloS one*, vol. 13, no. 9, e0203629, 2018.

[56] H. K. Walker, W. D. Hall, and J. W. Hurst, "Clinical methods: The history, physical, and laboratory examinations," 1990.

[57] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, Soviet Union, vol. 10, 1966, pp. 707–710.

Appendix

A Descriptive Statistics of Fixation Patterns, IDE Feature Usage, and Code Generation Sources in the Study

Table 4. The Top-5 attention-focused tokens and AST types (PSI) measured by fixation time.

AST Type	Fixation Time (s)
identifier	114.7
comment	65.72
literal	27.03
keyword	14.78
rbracket	3.77

2-Gram of AST Type	Count
identifier, identifier	3702
comment, comment	2220
comment, identifier	899
identifier, comment	862
identifier, literal	651

Table 5. Count of IDE features usage in the three programming tasks accumulated from all users' data.

Category	Feature	Kakaroma	Calculator	ZooSystem	All
IDE Support Edit	Copy/Cut	60	19	65	144
	Paste/Duplicate/Multiple	38	16	56	110
	$Undo/$Redo	18	31	47	96
	CommentByLineComment	1	3	3	7
Run	Run/RunClass/Rerun	27	91	24	142
	Stop	14	18	0	32
Debug	Debug/DebugClass	5	1	0	6
	StepOver/Into/Out	9	0	0	9
	ToggleLineBreakpoint	8	0	1	9
	Resume	19	0	0	19
Find & Replace	ChooseLookup/ReplaceItem	12	16	28	56
	Previous/NextWord	4	8	0	12
	Find/SearchEverywhere	0	5	9	14
Code Inspection	GotoDeclaration	12	8	15	35
File	SaveAll	4	7	16	27
Copilot Usage	copilot.applyInlays	41	183	78	302

Table 6. Analysis of code generated sources: Copilot, IDE support features (e.g., paste), programmer typing.

Metric	Source	Kakaroma	Calculator	ZooSystem
# Usages of	Copilot	4.33	25.71	8.11
	IDE Support	4.22	1.86	6
	Typing	30	39.86	22.67
# Characters from	Copilot	332.78	1038.29	395.44
	IDE Support	26.56	7.57	195.78
	Typing	108.22	115.14	96.78
% Most Characters from	Copilot	77.8	66.7	66.7
	IDE Support	0	0	11.1
	Typing	22.2	33.3	22.2