Developer Behaviors in Validating and Repairing LLM-Generated Code Using IDE and Eye Tracking

Ningzhi Tang*†, Meng Chen*†, Zheng Ning†, Aakash Bansal†, Yu Huang‡, Collin McMillan†, Toby Jia-Jun Li†
†{ntang, mchen24, zning, abansal1, cmc, toby.j.li}@nd.edu, ‡yu.huang@vanderbilt.edu
†University of Notre Dame, Notre Dame, IN, USA ‡Vanderbilt University, Nashville, TN, USA

2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) | 979-8-3503-6613-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/VL/HCC60511.2024.00015

***Abstract*—The increasing use of large language model (LLM)-powered code generation tools, such as GitHub Copilot, is transforming software engineering practices. This paper investigates how developers validate and repair code generated by Copilot and examines the impact of code provenance awareness during these processes. We conducted a lab study with 28 participants tasked with validating and repairing Copilot-generated code in three software projects. Participants were randomly divided into two groups: one informed about the provenance of LLM-generated code and the other not. We collected data on IDE interactions, eye-tracking, cognitive workload assessments, and conducted semi-structured interviews. Our results indicate that, without explicit information, developers often fail to identify the LLM origin of the code. Developers exhibit LLM-specific behaviors such as frequent switching between code and comments, different attentional focus, and a tendency to delete and rewrite code. Being aware of the code's provenance led to improved performance, increased search efforts, more frequent Copilot usage, and higher cognitive workload. These findings enhance our understanding of developer interactions with LLM-generated code and inform the design of tools for effective human-LLM collaboration in software development.**

***Index Terms*—GitHub Copilot, developer behavior analysis, debugging strategy, eye tracking, IDE tracking**

I. Introduction

Recent advances in LLMs have shown the potential to increase developers' productivity [1], [2]. A notable example is GitHub Copilot [3], a code generation tool powered by OpenAI's GPT model [4], which can generate lines or subroutines of code based on existing code or natural language comments [5]. However, the quality of the generated code is not guaranteed [6]. Developers must spend significant time evaluating its correctness, fixing potential bugs, and integrating the code into the existing codebase [7].

Error discovery and repair strategies are crucial for successful human-AI interaction, guiding the development of underlying AI models and user interfaces [8]–[10]. For code generation, AI tools can generate different types of errors than humans [11], and unlike humans, AI cannot articulate the rationale behind its decisions. Therefore, the focus of attention and strategies that developers follow may differ from those used when working with human-written code. Although studies have explored how developers interact with such AI tools to generate code [1], [12], [13], and the quality and usability of the generated code [5], [14], the process of validating and repairing them is still not well understood.

*Both authors contributed equally to this research.

Furthermore, as LLM-generated code becomes increasingly integrated with existing codebases, it is important to consider its provenance (i.e., whether the code is LLM-generated or human-written). Previous research has revealed that awareness of code provenance impacts developers' behavior when interacting with code, even though they may not always be conscious of such biases [15]–[18]. In the context of debugging, developers' awareness of code provenance may also impact their mental models of what is more likely to go wrong and how they may go wrong, affecting their strategies and performance. Understanding these differences can help developers better comprehend LLM-powered code generation tools, leading to the design of tools that establish appropriate trust and support better collaboration.

Prior studies on developers' interactions with LLM-powered code generation tools have largely relied on subjective analyses, such as post-analysis of audio/video recordings [7], surveys [1], and interviews [12], [13]. These methods may suffer from recall and observer biases [19], [20], and fall short of reliably capturing developers' cognitive processes [21]. In response, recent research in software engineering has increasingly utilized eye tracking as an objective measure. Eye-tracking data can be used to infer their visual attention strategies [22], [23] and may correlate with their self-reported cognitive workload [24], [25]. This provides an important indicator of programmers' mental model [26], [27]. Previous research also suggests that developer interactions within the Integrated Development Environment (IDE) can complement eye-tracking data to comprehensively understand behaviors [28], [29]. Therefore, we employ a mixed-method approach, combining quantitative (i.e., eye tracking and IDE tracking) and qualitative (i.e., surveys and interviews) methods to understand developer behaviors in our study.

This paper addresses the gap in understanding how developers validate and repair LLM-generated code, specifically focusing on chunks of code (about tens of lines) produced by Copilot. We also examine the effects of informing developers about the code provenance on their validation and repair strategies. We conducted lab studies with 28 participants, tasking them with validating, repairing, and integrating Copilot-generated code into three representative software projects. This study investigated the following research questions.

- **RQ1.** What are developers' perceptions and strategies for validating and repairing LLM-generated code compared to human-written code?

979-8-3503-6613-6/24/$31.00 ©2024 IEEE
DOI 10.1109/VL/HCC60511.2024.00015

- **RQ2.** How does awareness of code provenance affect the validation and repair behaviors of developers?

We highlight the following findings of our study.

- Developers often do not recognize the provenance of code unless explicitly informed. LLM-generated code typically has more comments, different error types, and stronger detail but weaker logic performance. Developers display LLM-specific behaviors including frequent switching between code and comments, varied focus areas, and a tendency to delete and rewrite the code.
- Informing developers about the provenance of LLM-generated code positively influences their performance in validation and repair tasks. It leads to improved validation and repair outcomes, increased search efforts, and more frequent utilization of Copilot during these processes, but it also increases their cognitive workload.

II. Related Work

A. Empirical Studies of LLM Code Generation Tools

Recent advances in transformer-based [30] LLMs marked a significant breakthrough in code generation. Several production-level code generation tools, such as GitHub Copilot, OpenAI GPT-4 [31], and Tabnine [32], have been adopted by a growing number of developers [1]. Previous studies explored how developers perceive and interact with these tools [12], [33]–[35]. For example, Barke *et al.* [13] observed two modes of developer interactions: *acceleration*, where developers use Copilot to code faster, and *exploration*, where developers use it to explore what to do next. Liang *et al.* [1] conducted a large-scale survey with 410 developers and identified a set of usage characteristics and usability issues with AI programming assistants. Our study focuses on how developers validate and repair Copilot-generated code, serving as a complementary perspective to the existing literature.

B. Debugging Strategies for Software Developers

Debugging is a complex process involving various tasks like code comprehension, error localization, and repair, and remains a long-standing topic in software engineering research [36]–[38]. Previous literature indicates that debugging involves iteratively comprehending the code and forming and testing hypotheses [39], [40]. Liu *et al.* [41] observed that common debugging techniques for large-scale software systems generally fall into two modes: identify and fix the error. However, LLM-generated code shows different traits from the human-written code. Previous studies found it to be less compact [42], lacking context [43], and of lower quality than human-written code [44]. Unlike previous studies, we investigate how developers validate and repair LLM-generated code. This helps us understand LLM's impact on software development and develop better tools for new challenges.

C. Eye Tracking in Software Engineering

Eye tracking captures participants' subconscious visual attention by recording their eye gaze data, reducing the subjectivity of self-report measures, e.g., think-aloud [45]. Recently, researchers started using it to study the software development process, including code comprehension [27], review [18], and debugging [28]. Eye tracking can also typically be integrated with tracking developers' interactions within the IDE to understand how they develop software [28], [46], [47]. In this study, we use CodeGRITS [48] to simultaneously track developers' IDE interactions and eye gazes for analysis.

Previous studies also used developer behavioral data (e.g., medical imaging and eye tracking data) to explore differences in software development behavior under different code provenance awareness, such as gender [15], [17] or human/machine distinctions [17], [18]. Our research extends these studies by using IDE and eye tracking to explore the impact of code provenance on validating and repairing LLM-generated code.

III. Study Design

A. Programming Tasks

In our study, each participant completed three Java programming tasks in different software engineering scenarios: algorithm design, graphical user interface (GUI) design, and object-oriented programming (OOP). For each task, we created a project containing several declared but unimplemented subroutines (e.g., classes/methods). Tasks 1 and 3 were adapted from undergraduate computer science assignments [49], [50].

We then developed prompts describing each subroutine's intended functionality and used Copilot to generate code. We used Copilot as a representative LLM-based code generation tool because it is specifically designed for coding, widely used [1], and previously studied in other works [7], [12].

- **Task 1.** *Kakamora (algorithm design)*: A LeetCode-like task that utilizes dynamic programming to find a path in a square array with the minimum sum. The entire program was written from scratch by Copilot.
- **Task 2.** *Calculator (GUI)*: A calculator app with a text interface and operational buttons, written using the Java GUI API. The code for GUI layout and button listeners was generated by Copilot.
- **Task 3.** *ZooSystem (OOP)*: A zoo management system with various animal classes with inheritance relationships (e.g., `Animal-Mammal-Lion`). Several management functions, such as add, delete, search, and display animals, were generated by Copilot.

We deliberately constructed prompts for which Copilot would generate code containing representative errors of different types. We categorized these errors using the taxonomy presented by Beizer [51]. The error statistics for the three tasks are presented in Table I in the Appendix. Each subtask refers to a subroutine generated by Copilot that contains bugs, which may include multiple error types (e.g., subtasks 1.2 and 3.2).

B. Participants

We recruited 28 participants (17 male, 11 female) from the local university community. Among them, 10 are graduate students and 14 are undergraduate students; 22 major in computer science and engineering. Three participants had no prior programming experience in Java, but they had six,

seven, and 12 years of experience in other programming languages, respectively. 12 had just completed an introductory course on Java programming, 12 had one to three years of Java programming experience, and one had over three years. Regarding their general programming experience, participants on average had 5.5 years, and only three of them had less than three years of experience. Eight of them had used Copilot in the past before our study. The participants received a $30 Amazon Gift Card as compensation for their time.

C. Study Protocol

We conducted the study in person in a usability lab using a Windows 10 computer with a 27-inch monitor and a Tobii Pro Fusion eye tracker [52], which sampled at 60 Hz. Participants used IntelliJ IDEA 2022.1.4 with the Copilot and CodeGRITS [48] plugins for IDE tracking, eye tracking, and screen recording. The IDE font size was set to 20 points. 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.

We randomly assigned the participants into two groups, each consisting of 14 participants, with different information about the code provenance. The **Informed** group was explicitly told that the code was generated by Copilot, while the **Non-Informed** group received no such indication.

Each study session lasted approximately two hours. After the participants signed the consent form and completed a pre-study demographic questionnaire, we briefed them about the overall objective and process of the study, followed by a 10-minute tutorial on using IntelliJ IDEA and Copilot. We also gave instructions on interacting with the eye tracker (e.g., refraining from major head movements).

For each task, participants first read an instructional document about the task background and then calibrated the eye tracker. They had 20 minutes to complete each task without browsing the Internet. After each task, participants completed a NASA Task Load Index (NASA-TLX) questionnaire [53] to self-report their perceived workload across six dimensions: Mental Demand, Physical Demand, Temporal Demand, Performance, Effort, and Frustration.

At the end of the study, we conducted a 20-minute semi-structured interview with participants. We discussed their perceptions of LLM-generated code quality, perceived differences between LLM-generated and human-written code, strategies for validating and repairing LLM-generated code, use of IDE features and Copilot, and changes in mental models and strategies. We also asked the Non-Informed group if they realized the code was LLM-generated during the study.

D. Threats to Validity

Our study faces several validity threats, one of which involves the programming task selection. We chose three lightweight tasks in a popular programming language (i.e., Java), designed to represent typical types of programming tasks, and iteratively invoked Copilot to generate code with representative error types. Despite this, these tasks may not fully capture the complexity and diversity of real-world programming. Then, participant selection poses another threat. Most participants were Computer Science students, which may not represent professional developers' experiences. To enhance ecological validity, a future longitudinal study in actual software development settings would be beneficial. Furthermore, eye tracking often shows drift after prolonged use [54]. To mitigate this, we performed calibration before each task, used well-documented eye-tracking metrics and analyses [55], and supplemented it with self-reported workload via NASA-TLX [53].

IV. Research Methods

A. Quantitative Analysis

1) Gaze Pattern Metrics: Fixation refers to stabilization of the eye at one location for a period of time, which is commonly used in eye-tracking studies and is often considered to be associated with visual attention and cognitive workload [56], [57]. Following the practice in previous research [55], we identified fixations by extracting gazes on the same token with durations longer than 200 ms, and we considered the transitions between two fixations within 50 ms as saccades. We computed the following metrics using the definitions provided in [55] to analyze the cognitive processes of the participants.

- *Fixation Time*: Total duration of all fixations in seconds.
- *Fixation Count*: Total number of fixations.
- *Average Fixation Duration*: Average duration of each fixation in seconds.
- *Saccade Time*: Total duration of all saccades in seconds.
- *Saccade Count*: Total number of saccades.
- *Average Saccade Length*: Average length of each saccade in pixels.

2) Developer Behavior Categorization: For eye tracking data, based on the fixations, we classified the tokens that the developer looks at into three types: code, comment, and document. This classification allowed us to identify three types of reading behaviors (Behavior 1-3 Table II in the Appendix). For IDE interaction data, an author categorized them according to the similarity of their underlying activity types into initial high-level behaviors (e.g., `Debug`, `StepOver`, and `ToggleLineBreakpoint` into "Employing Debugger") and discarded data that cannot be meaningfully aggregated (e.g., `ToggleInsertState`). Subsequently, another author reviewed and, if necessary, revised the categorizations and definitions of behaviors. The disagreement cases were discussed to reconcile differences and establish a cohesive categorization (Behavior 4-11 in Table II in the Appendix).

B. Qualitative Analysis

For the qualitative analysis of interview transcripts, we followed established open-coding procedures [58], [59]. Initially, two authors independently performed qualitative coding on the transcripts. Subsequently, they discussed their findings to achieve agreement and formed a consolidated codebook. Using this codebook, we conducted a thematic analysis to identify emerging themes from the interviews. These themes were refined and developed into study findings.

V. Study Result

In this section, we report the results for our RQs. Except for Fig. 1, all calculations of frequency or time were averaged across each task for all participants.

A. RQ1. Perceptions and Strategies of Developers

1) Perceptions of LLM-Generated Code: We asked the fourteen participants from the Non-Informed group at the beginning of the interview, "*During the tasks, did you realize that the code you just validated was actually generated by LLM?*" Almost 80% (11/14) of them reported they did not realize it. This indicates that if not clearly informed, developers may not consider the provenance of the code, or they may implicitly assume that the code was written by a human. However, different perceptions of the code provenance impact validation and repair strategies, workload, and performance, which is further discussed in Section V-B.

Many participants (11 out of 28) indicated that the code had a good coding style and readability, even better than human-written code. For example, P21 reported, "*[LLM-generated code] does follow human formatting guidelines; variable names and everything were verbose and easy to use.*" (P10) P14, P15, P17, P21, P23, and P25 highlighted that LLM-generated code shows better understandability as it contains more detailed comments compared to human-written code.

However, participants reported that LLM made some mistakes that human developers would not make. For example, in Task 1, Copilot used a series of if statements to hard-code the test samples. P19, P21, P24, and P26 all expressed confusion: "*When I saw the hardcoded '1', '2', and '3' in the path reconstruction, I was really confused and thought there would be some special meanings.*" (P21).

P6, P16, P25, and P27 also mentioned that LLM-generated code generally avoids detailed errors like syntax bugs, but its semantics and behaviors often misalign with user intents. "*Humans are more error-prone than Copilot when it comes to details; for the logic [of code], I think Copilot is more error-prone*". (P25) This also affects their validation and repair approaches, which is discussed more in Section V-A2.

2) Strategies to Validate and Repair Code: Combining the findings of the interviews with tracked data, we identified the following findings about developers' validation and repair strategies specific to LLM-generated code.

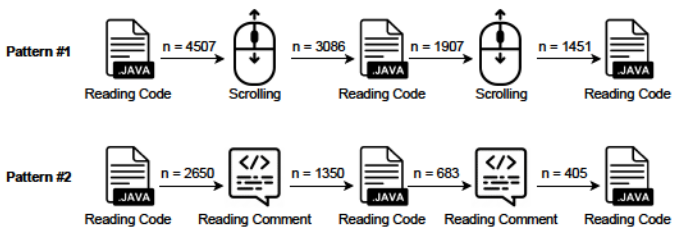

Fig. 1. Top-2 most frequent behavior transition patterns across all participants.

First, based on developer behavior categorization in Section IV-A2, we selected the two most frequent patterns of behavior transition sequences in Fig. 1. Pattern #2 shows developers frequently switch fixations between code and comments. Since the comments in the task projects are used as prompts for Copilot to generate buggy code, we interpret this as that developers are trying to "*disambiguate the mismatch between the code and the prompt.*" (P14) P3 also reported, "*Switching between instructions and code is annoying and challenging.*", consistent with other studies [60], [61].

Second, some LLM-generated code contains numerous erroneous statements that are difficult to fix by modifying only small parts. Participants tend to delete or comment out almost all of them and rewrite the code themselves. For example, P17 stated, "*With the layout, I just completely threw out the existing layout and kind of made one that worked myself.*"

Third, as stated in Section V-A1, participants think that LLM is more likely to make errors in the overall logic, while humans are more likely to make mistakes in the details. For example, P25 states that "*I will focus more on the logic of the code if it is generated by LLM, probably because I don't think Copilot can generate really complicated logic*"; LLM rarely makes mistakes in generating similar, repetitive code blocks – "*If you use LLM to generate similar code, it tends to be either all correct or all incorrect.*" Therefore, "*If it is a similar bunch of code, I would trust the LLM; as for logic details, I will double-check the correctness of the LLM-generated code.*"

Fourth, we allowed participants to use Copilot in our study. They reported that Copilot itself facilitated the process of idea exploration, and provided syntax suggestions while accelerating coding, consistent with previous research [1], [13]. On average, participants used Copilot to generate more characters than by typing keystrokes (362.84 $>$ 131.31, Student's t-test, p-value = 0.0007 $<$ 0.01). Notably, we also observed that participants used Copilot to generate inline comments to help understand the generated code. P25 states, "*I think if that line of code has a bug, generating the comments directly from the code will help me figure it out.*"

Key findings: If uninformed, developers may not recognize the code is LLM-generated. LLM-generated code performs well in terms of coding style and readability. However, it tends to make mistakes uncommon for human developers. Developers exhibit several behaviors specific to handling LLM-generated code: they frequently switch between code and comments, often completely delete and rewrite code, and exhibit a shifted focus from syntax to logic. Copilot is frequently used to enhance the process, particularly by generating inline comments.

B. RQ2. Effects of Code Provenance Knowledge

In this section, we investigate the impact of code provenance, i.e., whether the code was written by LLM or humans, on developer behaviors.

1) Bug Fixing Success Rate: We first analyze bug-fixing success rates in the study (see bugs details in Table I in the Appendix). As shown in Fig. 2, we observed that the Informed group performs better than the Non-Informed group (average

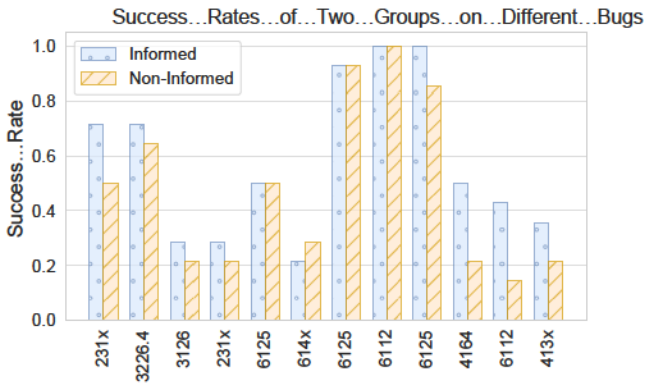

Fig. 2. The success rates of bug validating and repair of the two groups.

success rate 0.577 > 0.446, Student's t-test, p-value = 0.0417 < 0.05). When participants were informed that the code was written by Copilot, they performed better on 8 bugs, the same on 3 bugs, and worse on 1 bug.

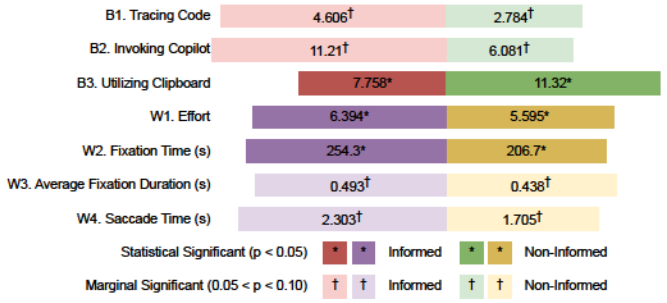

Fig. 3. Comparison of developer behavior frequencies (B1-B3), as well as self-reported workload measured by NASA-TLX and eye-tracking-related metrics (W1-W4) between the two groups. Only (marginally) statistically significant results from the Student's t-test are reported.

2) Developer Behaviors: As stated in Section V-A2, participants. Thus, we first computed the ratio of time spent on code to comments to see if there were any differences between the two groups. These metrics can be inferred from eye-tracking data. The results suggest that the Informed group had a higher, which is partially significant (0.233 > 0.179, Student's t-test, p-value = 0.093 < 0.10). The Pearson correlation analysis also showed that the higher ratio was positively correlated with the success rates across 12 bugs for each person (0.103 with a p-value of 0.066 < 0.10). Additionally, the Informed group showed significantly less time reading the original document (48.6 < 67.4, Student's t-test, p-value = 0.029 < 0.05).

As shown in Fig. 3, we calculated the frequency of developer behaviors, excluding reading, for both groups (see details in Behaviors 4-11 of Table II in the Appendix). We make two observations for the (partially) statistically significant results: *First*, if informed that the code was generated by LLM, participants use tracing code features (e.g., `Find`, `GotoDeclaration`, `FindUsages`) more frequently (B1). Fig. 3 also shows that they have higher Saccade Time (W4), which indicates greater search effort [62]. This may be because they focus more on the high-level logic of the code than on the low-level details, as stated in Section V-A2. *Second*, if informed that the code is generated by LLM, participants use Copilot more (B2) and the clipboard less (B3).

3) Cognitive Workload: We also calculated the workload metrics based on NASA-TLX [53] and the gaze pattern metrics from Section IV-A1. Fig. 3 reveals that the Informed group had higher self-reported effort (W1), fixation time (W2), and average fixation duration (W3). Longer fixations indicate greater visual attention and cognitive workload [56], [57]. This suggests that being aware of LLM-generated code's provenance increases cognitive workload.

Key findings: When developers are informed that the code is generated by LLM, they perform better in validating and repairing the code. Subsequently, they spend more time examining the prompts provided to Copilot compared to the code, and spend less time reading documents. When informed, developers make a greater search effort by using tracing code features more frequently and exhibit higher saccade times. They also used Copilot more frequently and used the clipboard less frequently. Finally, they experience a higher cognitive workload, as indicated by self-reported effort, fixation time, and average fixation duration.

VI. Conclusion and Future Work

We conducted a lab study with 28 participants to observe their behavior while validating and repairing LLM-generated code. Our findings indicate that, without explicit notification, developers often fail to recognize the code's provenance, which can impact their performance, behaviors, and cognitive workload. Although the LLM-generated code exhibits good readability and style, it introduces atypical errors. Developers also demonstrate unique behaviors in this context.

These observations underscore the need for new tools and methods tailored to support developers in the LLM era. For example, increasing awareness of code provenance may enhance their abilities, likely due to a more focused approach to generation prompts and a better understanding of potential issues to anticipate. Additionally, the frequent switching between code and comments/prompts suggests a need for improved interfaces in development tools. Visualization tools that connect prompts with generated code could reduce switching costs and enhance developers' understanding and efficiency.

In the future, we plan to systematically investigate the characteristics of LLM-generated code from a human perception perspective, complementing our behavior-centric approach. This effort will inform the development of more effective tools for developers working with LLM-generated code. Additionally, we plan to expand our research to include professional developers in long-term, real-world studies. We also intend to incorporate additional biometric sensors, such as heart rate monitors and fMRI, to gain a better understanding of the cognitive processes involved in software development.

ACKNOWLEDGMENT

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 Gelei Xu, Junwen An, and Chaoran Chen for useful discussion and valuable feedback on the project. We also thank Robert Wallace for his assistance in setting up the eye tracker.

REFERENCES

[1] J. T. Liang, C. Yang, and B. A. Myers, "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*, 2024, pp. 1–13.

[2] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," *arXiv preprint arXiv:2302.06590*, 2023.

[3] "Github copilot · your ai pair programmer," retrieved July 1, 2024 from https://github.com/features/copilot/.

[4] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[6] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, 2023.

[7] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "Reading between the lines: Modeling user behavior and costs in ai-assisted programming," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–16.

[8] 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.

[9] 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.

[10] 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.

[11] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "Github copilot ai pair programmer: Asset or liability?" *Journal of Systems and Software*, vol. 203, p. 111734, 2023.

[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," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.

[14] N. Al Madi, "How readable is model-generated code? examining readability and visual inspection of github copilot," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[15] J. Terrell, A. Kofink, J. Middleton, C. Rainear, E. Murphy-Hill, C. Parnin, and J. Stallings, "Gender differences and bias in open source: Pull request acceptance of women versus men," *PeerJ Computer Science*, vol. 3, p. e111, 2017.

[16] N. Imtiaz, J. Middleton, J. Chakraborty, N. Robson, G. Bai, and E. Murphy-Hill, "Investigating the effects of gender bias on github," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 700–711.

[17] Y. Huang, K. Leach, Z. Sharafi, N. McKay, T. Santander, and W. Weimer, "Biases and differences in code review using medical imaging and eye-tracking: genders, humans, and machines," in *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 456–468.

[18] I. Bertram, J. Hong, Y. Huang, W. Weimer, and Z. Sharafi, "Trustworthiness perceptions in code review: An eye-tracking study," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–6.

[19] A. Eteläpelto, "Metacognition and the expertise of computer program comprehension," *Scandinavian Journal of Educational Research*, vol. 37, no. 3, pp. 243–254, 1993.

[20] M. C. Davis, E. Aghayi, T. D. Latoza, X. Wang, B. A. Myers, and J. Sunshine, "What's (not) working in programmer user studies?" *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–32, 2023.

[21] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 177–187.

[22] R. Bednarik and M. Tukiainen, "An eye-tracking methodology for characterizing program comprehension processes," in *Proceedings of the 2006 symposium on Eye tracking research & applications*, 2006, pp. 125–132.

[23] C. Aschwanden and M. Crosby, "Code scanning patterns in program comprehension," in *Proceedings of the 39th hawaii international conference on system sciences*. Citeseer, 2006.

[24] O. Palinko, A. L. Kun, A. Shyrokov, and P. Heeman, "Estimating cognitive load using remote eye tracking in a driving simulator," in *Proceedings of the 2010 symposium on eye-tracking research & applications*, 2010, pp. 141–144.

[25] 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.

[26] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th international conference on Software engineering*, 2014, pp. 390–401.

[27] A. Bansal, R. Wallace, Z. Karas, N. Tang, Y. Huang, T. J.-J. Li, and C. McMillan, "Programmer visual attention during context-aware code summarization," *arXiv preprint arXiv:2405.18573*, 2024.

[28] 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.

[29] M. Kazemitabaar, X. Hou, A. Henley, B. J. Ericson, D. Weintrop, and T. Grossman, "How novices use llm-based code generators to solve cs1 coding tasks in a self-paced learning environment," in *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, 2023, pp. 1–12.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[31] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[32] "Tabnine ai code assistant — private, personalized, protected," retrieved July 1, 2024 from https://www.tabnine.com/.

[33] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 21–29.

[34] T. Wu, K. Koedinger *et al.*, "Is ai the better programming partner? human-human pair programming vs. human-ai pair programming," *arXiv preprint arXiv:2306.05153*, 2023.

[35] M. Amoozadeh, D. Daniels, D. Nam, A. Kumar, S. Chen, M. Hilton, S. Srinivasa Ragavan, and M. A. Alipour, "Trust in generative ai among students: An exploratory study," in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, 2024, pp. 67–73.
[36] 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.
[37] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 185–194.
[38] A. Alaboudi and T. D. LaToza, "What constitutes debugging? an exploratory study of debugging episodes," *Empirical Software Engineering*, vol. 28, no. 5, p. 117, 2023.
[39] K. Araki, Z. Furukawa, and J. Cheng, "A general framework for debugging," *IEEE software*, vol. 8, no. 3, pp. 14–20, 1991.
[40] D. J. Gilmore, "Models of debugging," *Acta psychologica*, vol. 78, no. 1-3, pp. 151–172, 1991.
[41] A. Liu and M. Coblenz, "Debugging techniques in professional programming." Plateau Workshop.
[42] 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.
[43] Y. Li, Y. Peng, Y. Huo, and M. R. Lyu, "Enhancing llm-based coding tools through native integration of ide-derived static context," *arXiv preprint arXiv:2402.03630*, 2024.
[44] 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.
[45] M. A. Just and P. A. Carpenter, "A theory of reading: from eye fixations to comprehension." *Psychological review*, vol. 87, no. 4, p. 329, 1980.
[46] 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.
[47] S. Maan, "Representational learning approach for predicting developer expertise using eye movements," 2020.
[48] N. Tang, J. An, M. Chen, A. Bansal, Y. Huang, C. McMillan, and T. J.-J. Li, "Codegrits: A research toolkit for developer behavior and eye tracking in ide," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 119–123.
[49] "Challenge 12: Kakamora," retrieved July 1, 2024 from https://www3.nd.edu/ pbui/teaching/cse.30872.fa22/challenge12.html.
[50] "Programming paradigms summer 2022 - nd," retrieved July 1, 2024 from https://www3.nd.edu/ skumar5/teaching/2022-summer-pp.html.
[51] B. Beizer, *Software testing techniques*. Dreamtech Press, 2003.
[52] "Reach further with your research — tobii pro fusion," retrieved July 1, 2024 from https://www.tobii.com/products/eye-trackers/screen-based/tobii-pro-fusion.
[53] S. G. Hart, "Nasa task load index (tlx)," 1986.
[54] Z. Sharafi, B. Sharif, Y.-G. Guéhéneuc, A. Begel, R. Bednarik, and M. Crosby, "A practical guide on conducting eye tracking studies in software engineering," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3128–3174, 2020.
[55] 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.
[56] M. Dorr, T. Martinetz, K. R. Gegenfurtner, and E. Barth, "Variability of eye movements when viewing dynamic natural scenes," *Journal of vision*, vol. 10, no. 10, pp. 28–28, 2010.
[57] H. Sheridan and E. M. Reingold, "Chess players' eye movements reveal rapid recognition of complex visual patterns: Evidence from a chess-related visual search task," *Journal of vision*, vol. 17, no. 3, pp. 4–4, 2017.
[58] M. Brod, L. E. Tesler, and T. L. Christensen, "Qualitative research and content validity: developing best practices based on science and experience," *Quality of life research*, vol. 18, pp. 1263–1278, 2009.
[59] J. Lazar, J. H. Feng, and H. Hochheiser, *Research methods in human-computer interaction*. Morgan Kaufmann, 2017.
[60] A. Sarkar, A. D. Gordon, C. Negreanu, C. Poelitz, S. S. Ragavan, and B. Zorn, "What is it like to program with artificial intelligence?" *arXiv preprint arXiv:2208.06213*, 2022.
[61] R. Yen, J. Zhu, S. Suh, H. Xia, and J. Zhao, "Coladder: Supporting programmers with hierarchical code generation in multi-level abstraction," *arXiv preprint arXiv:2310.08699*, 2023.
[62] P. Alex, "Eye tracking in human-computer interaction and usability research: Current status and future prospects," *The Encyclopedia of Human Computer Interaction*, pp. 211–219, 2005.

APPENDIX

TABLE I
BUG TYPES REPRESENTED IN PROGRAMMING TASKS BASED ON [51].

Task	Subtask	Bug Index	Bug Category
Kakaroma	1.1	231x	Missing Case
		3226.4	String Manipulation-Insertion
	1.2	3126	Illogic Predicates
		231x	Missing Case
Calculator	2.1	6125	Parameter Value
	2.2	614x	Initialization State
ZooSystem	3.1	6125	Parameter Value
	3.2	6112	Wrong Component
		6125	Parameter Value
	3.3	4164	Should be Dynamic Resource
	3.4	6112	Wrong Component
	3.5	413x	Initial, Default Values

TABLE II
CATEGORIZATION OF DEVELOPER BEHAVIOR EMERGED FROM IDE AND EYE TRACKING DATA COLLECTED BY CODEGRITS [48].

Index	Behavior	Tracking Data
1	Reading Document	Consecutive fixations on the instructional document
2	Reading Code	Consecutive fixations on the code
3	Reading Comment	Consecutive fixations on the comment
4	Switching Files	Opening, closing, or changing the selection of a file
5	Scrolling	Scrolling a file via mouse wheel, arrow keys, or touchpad gestures
6	Tracing Code	Searching tokens, finding usages or going to declarations
7	Running for Output	Running the class to obtain execution output
8	Employing Debugger	Utilizing debugger and its corresponding features (e.g., toggling breakpoints)
9	Invoking Copilot	Accepting, rejecting, or browsing code generated from Copilot
10	Utilizing Clipboard	Copying, cutting, or pasting contents
11	Keystrokes Typing	Typing characters using keystrokes