GazeCopilot: Evaluating Novel Gaze-Informed Prompting for AI-Supported Code Comprehension and Readability

YASMINE ELFARES, University of Glasgow, United Kingdom
GÜL ÇALIKLI, University of Glasgow, United Kingdom
MOHAMED KHAMIS, University of Glasgow, United Kingdom

AI-powered coding assistants, like GitHub Copilot, are increasingly used to boost developers' productivity. However, their output quality hinges on the contextual richness of the prompts. Meanwhile, gaze behaviour carries rich cognitive information, providing insights into how developers process code. We leverage this in Real-time GazeCopilot, a novel approach that refines prompts using real-time gaze data to improve code comprehension and readability by integrating gaze metrics, like fixation patterns and pupil dilation, into prompts to adapt suggestions to developers' cognitive states. In a controlled lab study with 25 developers, we evaluated Real-time GazeCopilot against two baselines: Standard Copilot, which relies on text prompts provided by developers, and Pre-set GazeCopilot, which uses a hard-coded prompt that assumes developers' gaze metrics indicate they are struggling with all aspects of the code, allowing us to assess the impact of leveraging the developer's personal real-time gaze data. Our results show that prompts dynamically generated using developers' real-time gaze data significantly improve code comprehension accuracy, reduce comprehension time, and improve perceived readability compared to Standard Copilot. Our Real-time GazeCopilot approach selectively refactors only code aspects where gaze data indicate difficulty, outperforming the overgeneralized refactoring done by Pre-set GazeCopilot by avoiding revising code the developer already understands. **Data and Materials:** https://figshare.com/s/f08fec00e7b6e6badae1

1 Introduction

The rapid advancements in artificial intelligence (AI) have significantly transformed software development, with AI-assisted coding tools revolutionizing the way programmers write and interact with code. These AI-powered assistants generate code suggestions based on natural language prompts, improving developer productivity and efficiency. A key factor influencing the effectiveness of these tools is prompt engineering, as the quality of the prompts directly affects the accuracy and usefulness of the LLM generated code. It is equally important to not only generate functional but also readable and comprehensible code. Poor code readability can lead to increased cognitive load, difficulty in maintenance, and a higher likelihood of introducing errors [60].

Despite the efficiency gains provided by AI-assisted coding tools, two major problems persist. First, developers, especially novices, often struggle to craft effective prompts that produce clear and readable code [27, 45, 54]. Ineffective prompts can lead to ambiguous or overly complex AI-generated code [22], making it even harder for beginners to comprehend and refine their code, as they may not know how to improve it or identify specific issues to fix. Second, current AI models primarily optimize for functional correctness but do not explicitly consider how programmers cognitively process the generated code [43]. This gap results in AI-generated code that, while mostly syntactically correct, may not be easily readable, maintainable, or aligned with programmers' expectations [1].

Addressing these challenges is important as readable code is crucial for software maintainability, collaboration, and debugging. Code that is difficult to comprehend can slow down development, introduce errors, and increase technical debt [76]. As AI-assisted programming becomes more prevalent, ensuring that generated code meets readability standards is essential for fostering effective human-AI collaboration. Secondly, incorporating human cognitive insights and physiological data into AI-driven coding can potentially improve collaborative human-AI programming.

Authors' Contact Information: Yasmine Elfares, University of Glasgow, United Kingdom, y.elfares.1@research.gla.ac.uk; Gül Çalikli, University of Glasgow, United Kingdom, HandanGul.Calikli@glasgow.ac.uk; Mohamed Khamis, University of Glasgow, United Kingdom, Mohamed.Khamis@glasgow.ac.uk.

To this end, we propose Real-time GazeCopilot, a novel approach for integrating real-time eye-tracking data into prompt design. As gaze data reflects developers' comprehension and cognitive load through fixations, pupil dilation, and other gaze metrics [2], integrating it into prompts provides LLMs with richer context about the developer's mental state and actual understanding of the code. This has the potential to bridge the gap between developers' cognitive processing and prompt engineering, making AI-assisted coding more accessible and effective, especially for those who struggle to write effective prompts.

We then report on an empirical evaluation of Real-time GazeCopilot in a controlled lab study where 25 participants read three messy code snippets while their gaze data was recorded, then improved each snippet using one of three AI-assistants: **(i)** Standard Copilot (baseline), where participants crafted their own textual prompts, **(ii)** Real-time GazeCopilot, where the prompt was automatically generated based on the developer's ***real-time*** gaze data that indicated comprehension difficulty, and **(iii)** Pre-set GazeCopilot, where we used a ***fixed*** prompt to refactor the code. The Pre-set GazeCopilot treatment serves as a second control condition (besides Standard Copilot) and assumes that all of the developer's gaze metrics provided indications of high cognitive load and that the code required extensive refactoring without considering developers' actual real-time gaze data. This condition allowed us to isolate the effect of adding generic gaze-related information from that of developers' actual real-time gaze behaviour. To prevent systematic biases, we counterbalanced the order of AI assistant and code snippet assignments across participants. We measured participants' comprehension of the refactored code, their perceived code readability, and their experience in terms of perceived control, agency, and code ownership.

While our work is not the first to study gaze behaviour during programming, Real-time GazeCopilot is, to the best of our knowledge, the first system that integrates physiological data to improve prompts in software development contexts. Unlike existing AI-assisted coding solutions that rely solely on textual input, our approach incorporates real-time cognitive feedback from programmers as interpreted from their gaze data. This enables a novel interaction paradigm where AI outputs are shaped by insights about the developer's cognitive state.

Our results show that integrating real-time gaze data into prompt design significantly improves code comprehension and perceived readability. Participants rated Real-time GazeCopilot-refactored code as statistically significantly more readable than Standard Copilot-refactored code ($p_{\text{adj}} = .001$, mean rank = 1.44 vs. 2.60). Real-time GazeCopilot also yielded significantly higher comprehension accuracy than Standard Copilot ($p = .0122$) and than the Pre-set GazeCopilot ($p = .0137$).

Our proposed approach selectively refactors code aspects associated with comprehension difficulty inferred from developers' gaze behaviour. By focusing only on these code aspects, our method reduces the overhead of revising code that developers already understand, avoiding excessive refactoring with a pre-set prompt. This aligns with prior work showing that refactoring can be costly, risky, and context-dependent, and its benefits vary across different code regions [41], and that ad hoc or indiscriminate refactoring can be counterproductive [8]. These findings highlight the potential of real-time gaze data to meaningfully enhance prompt quality and improve human-AI collaboration in software development.

This research contributes to the broader goal of making AI tools more user-centric, improving both productivity and code quality in software development environments. Bridging cognitive feedback and AI prompt engineering opens new possibilities for more human-aware, adaptive coding assistants that go beyond purely text-based interactions. This could make AI-assisted programming more accessible, reduce cognitive load, and improve code quality and maintainability at scale. Beyond programming, it also demonstrates a method for integrating real-time physiological signals into AI systems, paving the way for more intuitive, context-aware human-AI collaborations.

2 Related Work

Our work is situated at the intersection of eye-tracking and AI-assisted programming. While prior research has explored each of these areas independently, integrating real-time gaze signals into LLM prompts remains underexplored.

AI-Assisted Coding and Prompt Engineering. AI coding tools, such as GitHub Copilot, can have a significant impact on aspects of developer productivity, including a decrease in development task time [86]. However, gains in developer speed due to AI assistance can be contingent on developer experience and task complexity, as confirmed by the controlled study by Paradis et al. [56], conducted at Google. Such performance variability across users further highlights the need for AI integration strategies that are adaptive to the needs of individual developers. The large-scale survey conducted by Sergeyuk et al. [65] showed that one significant barrier to the broader adoption of AI coding assistants is the challenge of communicating programmers' intentions to the AI assistant. Both studies indicate the need for context-sensitive user-centered AI integration strategies. To facilitate context-aware interactions, Ross et al. [63] introduced the Programmer's Assistant, a conversational interface grounded in the surrounding code context, enabling developers to engage in multi-turn dialogues with large language models (LLMs). However, the effectiveness of LLM-based tools relies on prompt formulation [49, 68] (e.g., providing sufficient context in the prompts) to articulate one's intent, which poses a challenge due to the imposed cognitive load [35], especially for novice programmers [48, 66, 83].

While previous work highlights the limitations of prompt quality that can lead to a loss of trust in AI-assisted coding, it overlooks how real-time cognitive signals can enhance AI in coding. We propose gaze-informed prompt engineering, which uses eye-tracking to adjust prompts based on user attention, enabling real-time, user-aligned code suggestions with minimal input. Such an approach is also a step towards eliminating the need for prompt engineering, which is a challenge to overcome, as Hassan et al. [35] point out, to effectively and efficiently leverage the complementary strengths of human developers and sophisticated AI systems.

Code Readability and Comprehension. Software developers spend a significant portion of their time reading code. Brooks et al. [7] argued that code should be written to minimize the time it takes someone else to understand it. In line with this foundational view, in the responses of the survey Ljung et al. [47] conducted, software developers endorsed the principles of Clean Code, associating them with improved maintainability and efficiency. The study also revealed that developers often prioritize speed and functionality in early iterations, refining readability in later stages. However, as Johnson et al. [37] argue, neglecting readability imposes cognitive effort on developers, which can hinder long-term code comprehension and slow down maintenance, ultimately creating productivity bottlenecks. Code comprehension is crucial for effective software maintenance.

The need to support code comprehension (e.g., improving code readability, providing information support) has significantly grown with the emergence of AI code assistants. Prather et al. [61] observed that many novice programmers misunderstood or unquestioningly accepted AI-generated code, leading to logic errors and poor outcomes. The study by Mailach et al. [48] also revealed that beginners frequently struggled to understand or adapt AI-generated code, resulting in low-quality or incomplete solutions, unless participants engaged in iterative refinement or requested clarifications. To aid code understanding, Nam et al. [53] developed GILT, an IDE plugin that leverages OpenAI's GPT-3.5-turbo to provide explanations for highlighted sections of code. While GILT focuses on providing code explanations to aid in code understanding, our approach prioritizes enhancing code readability. Similar to our approach, using GILT does not require writing prompts; instead developers can select from high-level request options (e.g., to get a summary of highlighted code, explanations on API calls). In contrast, our proposed approach adapts to users' comprehension as they interact with the code by leveraging gaze data as a real-time signal of cognitive load and attention.

Eye Tracking in Software Engineering. The integration of eye tracking into software engineering research has been advanced by recent methodological frameworks that highlight the value of gaze metrics for uncovering real-time comprehension patterns [32]. Eye-tracking has proven valuable for studying how developers interact with code, offering insights into debugging [46, 80], code summarization [39, 62], and problem-solving [82]. It reveals cognitive processes behind comprehension and how expertise shapes attention patterns [2, 5, 9, 51]. Gaze has also been explored in collaborative development. In remote pair programming, shared gaze improved referential clarity and joint focus [18]. Dual eye-tracking studies further show that gaze coupling indicates shared understanding and predicts collaborative success [77]. However, these studies primarily focus on human–human collaboration.

More recently, gaze has been applied to AI-assisted programming. Tang et al.[71] found that developers unaware of an LLM's involvement were more likely to overlook logic flaws, illustrating automation bias. Mohamed et al.[51] observed shifts in student attention during AI-assisted summarization. Al Haque et al. [33] combined gaze, EEG, and IDE logs to quantify cognitive load across AI and non-AI coding tasks. Other studies report reduced attention to AI suggestions [1, 52], with delayed verification increasing cognitive burden. These issues are especially pronounced for novices—Prather et al. [61] used gaze to uncover how students with low self-efficacy were more susceptible to misleading AI output and less able to catch their own mistakes.

However, most prior work treats gaze as a retrospective signal. Obaidellah et al.'s survey [55] emphasized its untapped potential for real-time interaction. Our work addresses this gap by treating gaze as an active input rather than a passive signal. We embed gaze into the prompts, allowing LLM outputs to adapt in real-time based on user attention and cognitive state—enhancing comprehension, readability, and the overall quality of AI assistance.

Table 1. Gaze metrics used by Real-time GazeCopilot pipeline and to generate prompt text. The fixed prompt used in the Pre-set GazeCopilot treatment also relies on the listed gaze metrics.

Gaze Metric†	Definition	Formula‡
Mean Fixation Duration	The average duration of fixations, where each fixation duration is the difference between the timestamps of the first and last gaze points within a fixation cluster.	$\frac{1}{N}\sum_{i=1}^{N}(\text{Time}_{\text{end},i} - \text{Time}_{\text{start},i})$
Mean Fixation Count per Second	Total number of fixations divided by the total gaze recording time in seconds	$\frac{\text{Total Fixations}}{\text{Total Time (ms)}/1000}$
Mean Saccade Length	The average Euclidean distance between successive gaze points, assuming the gaze coordinates are normalized and the screen width is 1920 pixels.	$\sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2} \times 1920\,\text{px}$
Mean Pupil Dilation	The average change in pupil diameter relative to a baseline. The baseline is calculated from samples in the first 60 milliseconds of the coding session.	$\frac{1}{M}\sum_{i=1}^{M}(\text{Pupil}_i - \text{Baseline})$

(†) Mean fixation duration is in milliseconds, Mean saccade length is in pixels, and Mean pupil dilation is in millimeters.
(‡) In the formulas, N is the number of total fixations and M is the total number of pupil dilations.

3 Real-time GazeCopilot: Concept and Implementation

To support the aims of this research, we developed Real-time GazeCopilot, a novel software pipeline that integrates real-time eye tracking with AI-assisted refactoring to improve code readability. Real-time GazeCopilot leverages the CodeGRITS toolkit [70], a JetBrains IDE plugin for software engineering research to enable the generation of context-aware prompts for GitHub Copilot based on developers' gaze behaviour while reading source code. The Real-time GazeCopilot software pipeline is publicly available in the replication package [4].

3.1 Prompt Design

(1) Selecting eye-tracking metrics. We selected eye-tracking metrics based on prior research highlighting their relevance in distinguishing how programmers interact with code. *Fixation duration*, *fixation count per second*, and

saccade length capture eye movement patterns that reflect the cognitive demands of readable versus poorly structured code [80], while *pupil dilation* is a well-established indicator of cognitive load during complex tasks such as code comprehension [2]. These metrics also help differentiate the gaze behaviour of novice and expert programmers, with novice patterns often indicating difficulty in comprehension and a greater need for assistance [2].

(2) Selecting eye-tracking thresholds. We employed *predefined thresholds* for each of the eye-tracking metrics as indicators that the developer may be struggling to comprehend the code. These thresholds are as follows: fixation duration = 241.31 ms, fixation count per second = 2.89, saccade length = 132.74 pixels, and relative pupil dilation = 0.1 mm. The threshold values were chosen based on mean values reported by Jensen et al. [80] on novice developers' gaze behavior with well-structured code [80]. We used these thresholds as reference points to indicate "high" levels of each eye-tracking metric when constructing the prompt, since our pilot tests showed that Copilot does not meaningfully interpret the raw numerical values of these metrics.

To confirm that our thresholds are suitable for the demographics of our recruited participants, we compared our participants' demographics with those of the participants in the study by Jensen et al. [80] and found them to be largely similar. The participants in the referenced study [80] were novice programmers with an average experience level of 2.35 ($SD = 0.51$) on a scale from 0 to 6. Similarly, the majority of our participants rated their programming experience compared to their peers as "average" ($n = 14$). This form of self-assessment has been empirically shown to be a reliable measure of programming experience, as demonstrated by Feigenspan et al. [26], whose model was also used by Jensen et al. [80]. This high similarity suggests that the thresholds chosen for our study are suitable for our recruited participants.

(3) Mapping eye-tracking metrics to cognitive states. We inferred programmers' cognitive states by their gaze data while they read code containing readability challenges (i.e., messy code), referring to previous research in the literature, as follows:

- **Mean fixation duration:** Longer fixation durations indicate sustained attention, deeper processing, and higher cognitive effort, often due to increased task complexity or ambiguity in the visual stimulus. This may be due to code complexity or the difficulty of detecting defects, requiring greater cognitive effort to process the given task [2, 10, 67].
- **Mean fixation count per second:** A higher fixation count typically suggests lower efficiency, as it reflects increased visual effort to locate relevant information within an area of interest (AOI) or stimulus [2]. It can also indicate that the layout or content requires more scanning, suggesting cognitive strain in locating meaningful cues [67].
- **Mean saccade length:** Shorter saccades are associated with novice behavior, while experts tend to make larger jumps between code blocks, as they can identify important sections and do not need to read the code line by line [2]. Prior research also shows that novice programmers read code more linearly compared to experts [9].
- **Pupil dilation:** Increased pupil dilation serves as a sensitive indicator of cognitive challenge, reflecting the mental effort required when working on difficult tasks [2, 29].

(4) Generating gaze-informed natural language prompt. We incorporate the eye-tracking metrics that exceed the predefined thresholds, along with the associated inferred cognitive states, into a natural language prompt. The resulting prompt text first describes the programmers' cognitive state (while reading the messy code) based on these metrics, followed by the command: "Improve the code." For instance, if two of the gaze metrics values (e.g., mean fixation duration and mean pupil dilation) exceed the predefined thresholds, we include these metrics along with the cognitive states associated with them as shown in Figure 1a.

Example Real-time GazeCopilot Prompt

While reading the code, the developer demonstrated long fixation durations, indicating sustained attention and deep cognitive processing, suggesting higher complexity or ambiguity in the code, increased pupil dilation reflecting high cognitive effort and mental workload. Improve the code.

(a) Example Real-time GazeCopilot prompt including only the gaze metrics whose values exceed the predefined thresholds. The gaze metrics with values exceeding the thresholds are fixation duration and pupil dilation in this example.

Pre-set GazeCopilot Prompt

While reading the code, the developer demonstrated long fixation durations indicating sustained attention and deep cognitive processing, suggesting higher complexity or ambiguity in the code, high fixation count suggesting low visual efficiency, increased scanning, suggesting cognitive strain in locating meaningful cues, short saccades indicating novice-like behavior and linear reading patterns, reflecting difficulty in identifying key code elements, increased pupil dilation reflecting high cognitive effort and mental workload. Improve the code.

(b) Pre-set GazeCopilot prompt assuming that all four gaze metrics values exceed predefined thresholds. In Real-time GazeCopilot, if values of all four gaze metrics exceed thresholds, the resulting prompt becomes identical to Pre-set GazeCopilot prompt.

Fig. 1. Comparison of gaze-informed prompts used in Real-time GazeCopilot and Pre-set GazeCopilot, showing the text provided to Copilot in each treatment.

(5) Pilot Testing of Prompt Design. To explore whether GitHub Copilot takes gaze metrics into account when included in the prompt, we conducted pilot tests prior to conducting the main study to assess whether Copilot's responses are influenced by the provided context about programmers' gaze behavior compared to generic instructions without the gaze data (e.g., "Improve the code").

For each gaze metric (see Table 1), we generated a prompt following a format similar to that shown in Figure 1a. However, each prompt included only information about the programmer's cognitive state relevant to that specific gaze metric. For example, the prompt we prepared for **mean fixation duration** was: "*While reading the code, the developer demonstrated **long fixation durations**, indicating sustained attention and deep cognitive processing, suggesting higher complexity or ambiguity in the code. Improve the code.*". We also experimented with different combinations of gaze metrics to determine their effectiveness. Each generated prompt—whether based on a single metric or a combination—was then fed separately to Copilot for evaluation.

The results of our pilot testing are below. They are based on **(i)** our own qualitative evaluation of the resulting responses, and **(ii)** an examination of Copilot's rationale behind the response when prompted: "Describe how each of the gaze metrics was interpreted and mapped to specific code changes." and "Explain how the gaze metric influenced your decision to refactor the code."

- To address **high fixation count**, Copilot suggests more *meaningful variable names* and adds *concise comments*, making the code more self-explanatory and allowing users to focus on the logic rather than decoding the syntax or structure.
- Copilot mitigates **long fixation durations** by using *descriptive* and *consistent* identifiers and by simplifying logic to reduce *duplicate or redundant code*, thereby lowering the cognitive load per line of code.

- Copilot responds to **short saccade lengths** by enforcing *consistent formatting* and *modular structure* by breaking down long methods into smaller, methods that have *single-responsibility*, facilitating the grasp of the relationships between code blocks.
- Copilot aims to reduce cognitive load indicated by **pupil dilation** by breaking down code logic, *adding comments*, and avoiding ambiguous or error-prone programming patterns, such as *deeply nested conditionals*, *unclear variable names*, or *overly complex expressions*.

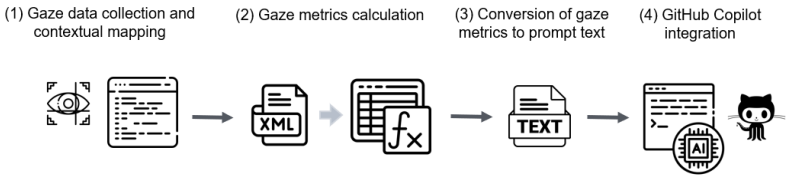

Fig. 2. Real-time GazeCopilot pipeline

3.2 Real-time GazeCopilot Pipeline

Figure 2 presents a high-level overview of the Real-time GazeCopilot pipeline, which we describe below.

(1) Gaze data collection and contextual mapping. We employed the *Tobii Pro Spark* [75] eye tracker, which allows for unobtrusive gaze capture during extended programming sessions, even with moderate head movements, since it is a screen-mounted (non-wearable) tracker that supports 60 Hz binocular tracking and automatic user detection. To interface with *Tobii Pro Spark*, we used the *Tobii Pro SDK for Python* [74], that provides real-time access to streamed gaze data including gaze coordinates, pupil diameter, gaze validity, and timestamps. *CodeGRITS* integrates with the Tobii Pro SDK, allowing us to capture both IDE interaction events (i.e., programmers' IDE interactions and screen recording) and gaze data in real-time, and synchronize them during the programming session. Once gaze data is streamed via the SDK, *CodeGRITS* maps each gaze point to the corresponding source code elements (i.e., file path, line, and column), synchronising it with IDE usage logs and screen recordings and creates a structured XML file with this mapping.

(2) Gaze metrics calculation. From the structured XML output generated by *CodeGRITS*, the pipeline calculates the four core eye tracking metrics described in Table 1: mean fixation duration, mean fixation count per second, mean saccade length, and mean pupil dilation.

(3) Conversion of gaze metrics to prompt text. The pipeline checks whether the calculated mean values of gaze metrics exceed the *predefined thresholds*. If a metric surpasses its threshold, it is included in the prompt along with the corresponding cognitive state (see Section 3.1).

(4) GitHub Copilot integration. We chose to feed the generated prompt text into GitHub Copilot [13] due to its widespread usage and seamless integration with modern IDEs. As of early 2025, over 15 million developers across more than 50,000 organizations were using GitHub Copilot [73]. Furthermore, GitHub Copilot integrates smoothly with IntelliJ IDEA via the official JetBrains plugin, aligning with our gaze data collection setup based on CodeGRITS.

The user interaction with GazeCopilot is similar to standard GitHub Copilot. Users trigger GazeCopilot via a keyboard shortcut, which displays a prompt generated based on their gaze behavior, allowing them to preview the suggested code.

4 Methodology - Evaluating Real-time GazeCopilot

The goal of this study is to investigate the impact of integrating real-time eye-tracking information into LLM prompts on code readability, comprehension and developers' experience in AI-assisted programming. All experiment artifacts are publicly available in the replication package [4].

4.1 Research Questions

We structure our study into two main research questions.

RQ1 How does incorporating *real-time* eye-tracking data into LLM prompts impact the readability and comprehension of AI-generated code?

RQ2 How does the inclusion of *real-time* eye tracking data into LLM prompts impact the developers' experience?

4.2 Experiment Design and Treatments

The study employs a within-subjects experimental design with one independent variable: Interaction Mode, which refers to the type of input prompt that is fed into GitHub Copilot. Each participant is exposed to all of the following three Interaction Modes:

- **Standard Copilot (Baseline).** Standard GitHub Copilot refactors the code based on the textual prompts the participants enter. This condition serves as a baseline representation of standard AI-assisted programming without additional enhancements.
- **Real-time GazeCopilot.** This treatment converts ***real-time*** eye tracking data of the participant into natural language text and adds it to the prompt and feeds it to Copilot. The wording depends on which eye-tracking metrics exceeded the thresholds (see an example in Figure 1a).
- **Pre-set GazeCopilot**. This treatment serves as another control (besides Standard Copilot), allowing us to examine whether the participants preferred general improvements to the refactored code or more adaptive adjustments based on their real-time gaze behavior. The pre-set prompt assumes that all four eye-tracking metrics exceeded thresholds and is fed to Copilot (see Figure 1b).

We counter-balanced the order of Interaction Modes and assigned a different code snippet to each mode to mitigate learning effects, resulting in six different sequences as shown in Table 2. We randomly assigned each participant to one of the six sequences.

Table 2. Experiment Design

	Code Snippets		
Sequence	**Snippet A**	**Snippet B**	**Snippet C**
Sequence 1	Standard Copilot	Real-time GazeCopilot	Pre-set GazeCopilot
Sequence 2	Standard Copilot	Pre-set GazeCopilot	Real-time GazeCopilot
Sequence 3	Real-time GazeCopilot	Standard Copilot	Pre-set GazeCopilot
Sequence 4	Real-time GazeCopilot	Pre-set GazeCopilot	Standard Copilot
Sequence 5	Pre-set GazeCopilot	Standard Copilot	Real-time GazeCopilot
Sequence 6	Pre-set GazeCopilot	Real-time GazeCopilot	Standard Copilot

Table 3. Code readability violations we injected in the code snippets to simulate realistic code issues in our experiment.

Readability Violation	Definition	Guidelines/Standards
Meaningless Naming	Classes, methods, and variables lack meaningful names, making the code difficult to interpret.	[28, 72, 81]
Inconsistent Naming	The code uses multiple naming styles inconsistently, making it harder to read and maintain.	[28]
Lack of Modular Structure	The system follows a monolithic design without a clear hierarchical structure or separation of concerns	[24, 28]
Lack of Structural Textual Features	The code lacks textual features that provide visual and structural cues, such as indentation, spacing and consistent formatting, which guide the reader through the code's logical flow.	[80]
Violation of the Single Responsibility Principle	The methods are excessively long and handle multiple tasks.	[17, 24, 28, 72, 81]
Lack of Comments and Documentation	The code has minimal or no comments explaining the logic and intent behind various components.	[17, 24, 28, 72, 81]
Duplicate/Redundant Code	(Nearly) identical code appears in multiple places instead of being refactored into reusable functions or classes.	[28]
High Cognitive Complexity	The code contains numerous control flow structures, nested logic and decision points that increase mental effort to follow the execution flow and reason about potential outcomes.	[12].
High Cyclomatic Complexity	The code contains multiple decision points, branching paths, and deeply nested loops and conditionals making it harder to analyze and predict execution flows	[28, 50, 72]

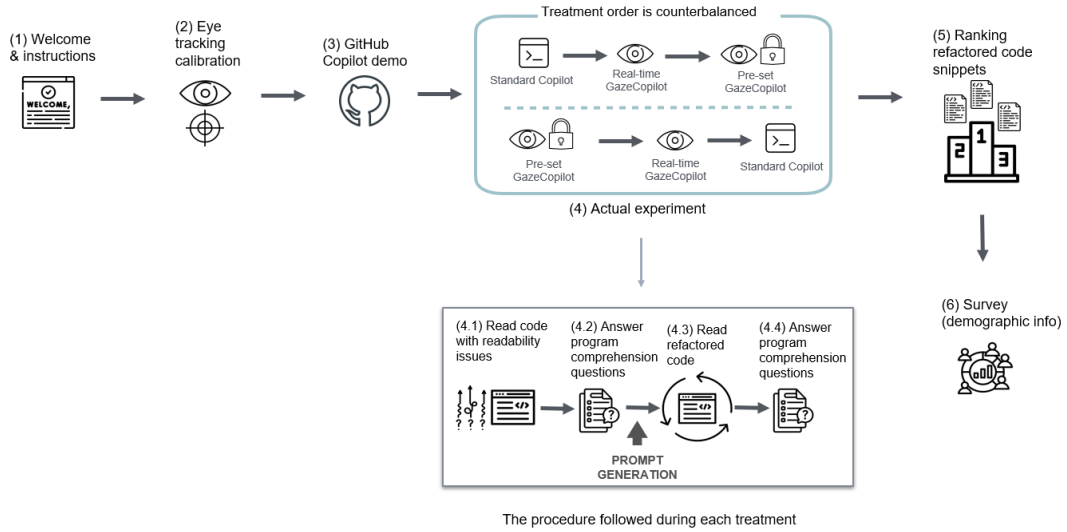

Fig. 3. Experiment flow

4.3 Experimental Procedure

We invited participants to our lab, where the lighting was controlled. We conducted the experiment with one participant at a time. The experiment stages are detailed below and in Figure 3.

(1) Welcome and instructions. Before the experiment began, participants received an information sheet detailing the study's purpose, tasks, and data collection methods. Participants provided informed consent before proceeding with the study. The study received ethical approval from our institution.

(2) Eye Tracker calibration. We calibrated the eye tracker for each participant using Tobii's default calibration procedure.

(3) GitHub Copilot demonstration. Prior to beginning the coding tasks, participants were given a brief hands-on demonstration of GitHub Copilot to ensure they understood how to interact with the tool. The demonstration showed how to trigger suggestions using the appropriate keyboard shortcuts and preview the generated code suggestions. This step was done to ensure that participants' familiarity with Copilot did not confound the experimental results or influence the evaluation of our system.

(4) Actual experiment. We assigned each participant to the three treatments (i.e., Standard Copilot, Real-time GazeCopilot, Pre-set GazeCopilot) in a counter-balanced order as mentioned in Section 4.2.

During each treatment (as shown in Figure 3), we asked participants to read a code snippet that has readability issues (i.e., messy code). The code was displayed using the IntelliJ IDE. While participants read the messy code (i.e., in stage (4.1)) their gaze was tracked to establish a *messy code reading behavior*. The participants then answered two Program Comprehension (PC) questions (stage (4.2)). The PC questions ask what the expected code output is, and for a high-level explanation of the code's functionality. We then asked participants to rate their confidence in their comprehension of the code.

The code was refactored using prompts in all treatments (stage (4.3) in Figure 3). In the Standard Copilot treatment, we instructed all participants: "You should use Github Copilot to generate an improved refactored version of the presented code. You may use however many prompts you need." All participants then created and submitted their own text prompts to Copilot. In the Real-time GazeCopilot treatment, the prompt was automatically generated from participants' real-time gaze data. In the Pre-set GazeCopilot, we used a pre-set (fixed) prompt that assumes all four eye-tracking metrics exceeded thresholds.

After refactoring, participants read the updated code, answered the same two PC questions about it, and indicated their confidence in their comprehension of the code (stage (4.4) in Figure 3).

Following the PC questions, participants answered questions about their subjective experience, including their percieved control, sense of agency, and percieved ownership of the code. While reading the refactored code (during stage (4.3)), we also tracked the participants' gaze to examine changes in gaze behaviour compared to the *messy code reading behavior*.

(5) Ranking refactored code snippets. Participants then ranked the three refactored code snippets from most to least readable and justified their choices.

(6) Demographics. Participants then completed demographic questions, including age, gender, and highest level of education, as well as programming experience. This information was required to help us identify the extent to which our participants represent novice developer population [23] and assess the suitability of the predefined gaze metric thresholds (see Section 3.1).

4.4 Experimental Objects

We used three Java code snippets that are **(i)** neither too trivial nor too complicated, **(ii)** self-contained, requiring no external dependencies, and **(iii)** do not rely on special technologies or frameworks/libraries.

To introduce readability challenges, we injected code readability violations based on coding standards and refactoring guidelines in the literature (see Table 3). To ensure similar comprehension difficulty, we used SonarQube [12] to calculate cognitive complexity scores, which reflect control flow and logic complexity. The messy snippets scored 50, 36, and 40, indicating comparable cognitive complexity. According to Campbell, who developed the cognitive complexity metric [12], a threshold of 15 is recommended [31]. All of our snippets exceed this threshold, suggesting they are sufficiently complex to require refactoring.

4.5 Variables and Measurements

The dependent variables are categorized into four sections:

Code Comprehension Metrics. Our analyses employ comprehension metrics based on previous research [57] that evaluate how source code readability impacts developers' ability to understand the code. Comprehension **accuracy** measures the correctness of participants' responses to the two PC questions, which is an ordinal value within the range $[0, 2]$. Additionally, we measure comprehension **time** as the time it takes to read the code snippet and answer the PC questions. We also capture participants' subjective **confidence** in their code comprehension using a 5-point Likert scale, where higher scores indicate greater confidence. Participants rated their confidence both before and after refactoring—that is, for the messy and refactored versions of each code snippet.

Code Readability Metrics. At the end of the actual experiment (stage (4) in Figure 3), we collect participants' **readability rankings** of the refactored versions of the three snippets from the most (1) to the least (3) readable. As an objective readability metric, we use **cognitive complexity** [12], which provides a readability-oriented code complexity measure. Cognitive complexity increases as the nesting of loops (`for`, `while`) and conditionals (`if-else`, `switch`) increases. Control structures such as loops, exception handling (`try-catch`), and boolean expressions with multiple logical operators (`&&`, `||`) contribute to an increase in cognitive complexity.

Eye-Tracking Metrics. We employ eye-tracking metrics (see Table 1) to assess gaze behaviour during the code comprehension tasks. We collect eye-tracking metrics for each participant during each Interaction Mode (i.e., Standard Copilot, Real-time GazeCopilot, and Pre-set GazeCopilot). This process is performed twice: once while participants read code with readability issues (i.e. messy code), and again while they read the refactored version.

User Perception and Agency. We evaluated participants' experience and sense of agency while using Standard Copilot, Real-time GazeCopilot, and Pre-set GazeCopilot, by conducting a separate post-task questionnaire after each Interaction Mode. We adapted the questionnaire questions from prior research on developer experience, with a particular focus on user agency and perceived system performance [36, 85]. The questionnaire consists of seven 5-point Likert scale questions where higher scores reflect more positive experiences or perceptions.

The experience questions comprise questions about **perceived understandability** (Q1), the need for **manual edits** (Q2), and code comprehension **speed** (Q7) of the refactored code, as well as the **intuitiveness** (Q6) of using each Interaction Mode. The sense of agency questions include the extent of **perceived control** over the refactoring process (Q3), **control satisfaction**—that is, satisfaction with the level of control (Q4)—and **code ownership**, or the extent to which the refactored code feels like one's own work (Q5).

4.6 Statistical Analyses

Having applied assumption checks[1], including Shapiro–Wilk test for normality of residuals and Mauchly's test for sphericity, we used parametric methods where data satisfied the assumptions and non-parametric methods otherwise.

To analyze **comprehension accuracy**, **time**, **confidence**, as well as **eye-tracking metrics**, we conducted Aligned Rank Transform (ART) ANOVA with fixed effects for Interaction Mode (Standard Copilot, Real-time GazeCopilot, and Pre-set GazeCopilot), version (messy / refactored), and their interaction, and participant as a random intercept to account for within-subject variability. To investigate significant effects further, we performed post-hoc pairwise comparisons using estimated marginal means with Bonferroni correction, based on the ART-aligned linear model. To

[1]The results of all assumption checks conducted are in the replication package in the file `supplementary_analyses_results.txt` under the folder `experiment`.

quantify the magnitude of observed differences (i.e., effect size), we calculated rank biserial (r_{rb}) correlations for each pairwise contrast using aligned rank-transformed data.

We used the Friedman test to analyze ordinal or non-parametric data—such as subjective **readability rankings**, the **cognitive complexity** scores of the refactored code snippets, and participants' responses to the seven **user perception and agency** questionnaire items. When the Friedman test indicated statistically significant differences, we reported Kendall's W as a measure of effect size. For post hoc comparisons, we used Wilcoxon signed-rank tests and calculated the corresponding rank biserial (r_{rb}) as an effect size estimate for each pairwise comparison.

4.7 Pilot Runs

We ran two pilot tests to **(i)** identify and address any technical issues, **(ii)** evaluate the snippets' complexity, **(iii)** assess the experiment duration, **(iv)** verify that the study instructions and interface are clear, and **(v)** collect feedback. Data from these pilot runs were excluded from the final analysis.

Following the first pilot, we revised the code snippets to ensure balanced and comparable complexity across all three tasks, and improved the questionnaire's wording for clarity. After the second pilot run, only minor revisions were required, and the experiment was deemed ready for the main study.

4.8 Sample Size

We conducted a series of *a priori* power analyses using G*Power [25] to estimate the minimum required sample size for detecting medium effects ($f = 0.25$) with $\alpha = 0.05$, power = 0.80, and three within-subject conditions, unless otherwise specified. We assumed sphericity ($\epsilon = 1$) and correlation $r = 0.6$. For nonparametric tests, Friedman and ART ANOVA, we used repeated-measures ANOVA as a conservative proxy for sample size estimation. Sample size estimation for repeated measures ANOVA yielded 23 participants. We recruited 25 participants, exceeding the required threshold to ensure adequate statistical power.

5 Results

5.1 Participant Demographics

Out of 25 participants, 16 (64%) reported having at least two years and 10 (40%) at least five years of programming experience. Overall, 15 participants (60%) reported programming regularly (40% weekly; 20% daily). When asked to rate their programming experience relative to their peers, 56% rated their experience as average, 28% as good, and 16% as poor.

Regarding GenAI Tools, 24% were familiar with GitHub Copilot, 68% had used other AI code assistants (e.g., ChatGPT, Claude, Gemini, and Cursor), and some had used both Copilot and other assistants or neither.

The sample included 14 participants who identified as female (56%) and 11 as male (44%). As we used eye tracking, we also collected vision information: All participants reported normal or corrected-to-normal vision (e.g., using glasses or contacts).

5.2 Impact of Gaze-enhanced Prompts on Code Readability and Comprehension (RQ1)

This section presents the results of how gaze-enhanced prompts impact code readability and comprehension.

Comprehension Accuracy. Figure 4 illustrates comprehension accuracy score changes across Standard Copilot, Real-time GazeCopilot, and Pre-set GazeCopilot Interaction Modes. It shows the number of participants whose

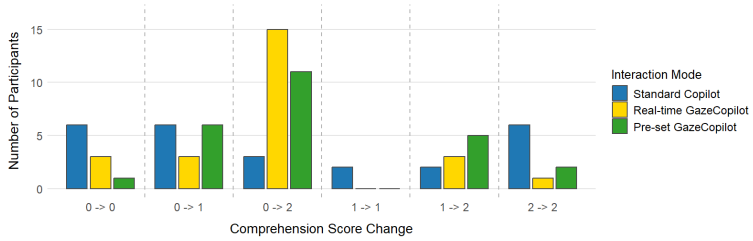

Fig. 4. Comprehension score changes across Interaction Modes: Standard Copilot, Real-time GazeCopilot, and Pre-set GazeCopilot (score range: 0–2, with higher scores indicating better comprehension). Each bar shows how many participants experienced a given score change $x \rightarrow y$ (e.g., $0 \rightarrow 1$ implies that the code comprehension score was 0 for messy code and improved to 1 for the refactored version). The Real-time GazeCopilot Interaction Mode showed the largest improvements.

comprehension scores changed after refactoring the messy code under each Interaction Mode. Notably, Real-time GazeCopilot yielded the largest gains, with 15 participants improving from a comprehension score of zero to a score of two (i.e., $0 \rightarrow 2$), compared to 11 in the Pre-set GazeCopilot and 3 in the Standard Copilot.

To account for participants who answered correctly before the code refactoring, we calculated the improvement in comprehension score by subtracting the comprehension score for the messy version (pre-refactoring) from the score for the refactored version. Descriptive statistics indicated the lowest mean improvement in the Standard Copilot condition ($M = 0.56$, $SD = 0.71$), with moderate gains in the Pre-set GazeCopilot Interaction Mode ($M = 1.32$, $SD = 0.69$), and the highest gains observed in the Real-time GazeCopilot condition ($M = 1.44$, $SD = 0.77$).

ART ANOVA results revealed a significant main effect of *version*, $F(1, 270) = 236.78$, $p < .001$, indicating that *version* has a significant impact on comprehension accuracy, with higher scores for *refactored* versions compared to the *messy* ones. We did not find statistically significant main effect of Interaction Mode, $F(2, 270) = 0.70$, $p = .496$; however, we found a significant interaction between Interaction Mode and version, $F(2, 270) = 11.46$, $p < .001$, indicating that the effect of Interaction Mode on comprehension accuracy depended on whether the code version was messy or refactored.

Post-hoc pairwise comparisons revealed that comprehension accuracy for Real-time GazeCopilot-refactored code was significantly higher than that of Standard Copilot-refactored code ($p = .0122$) with a moderate effect size (rank-biserial correlation $r_{rb} = 0.220$). Comprehension accuracy for Real-time GazeCopilot-refactored code was also significantly higher than that of Pre-set GazeCopilot-refactored code ($p = .0137$), with a large effect size ($r_{rb} = 0.698$). We did not find a significant difference between Standard Copilot-refactored and Pre-set GazeCopilot-refactored code ($p = 1.0000$).

In addition, post-hoc pairwise comparisons revealed a statistically significant improvement in code comprehension accuracy for Real-time GazeCopilot-refactored ($p = .0002$) and Pre-set GazeCopilot-refactored ($p = .0019$) code snippets compared to the corresponding messy versions.

To distinguish whether prior experience with GitHub Copilot had an impact on the improvement of comprehension accuracy, we ran additional tests. Participants **familiar** with Copilot ($n = 3$) showed mean comprehension gains of Standard Copilot = 0.33, Real-time GazeCopilot = 0.33, and Pre-set GazeCopilot = 1.33. Participants **unfamiliar** with Copilot ($n = 22$) showed corresponding gains of Standard Copilot = 0.59, Real-time GazeCopilot = 1.59, and

Pre-set GazeCopilot = 1.32. This shows that both groups benefited most from the Real-time GazeCopilot treatment, regardless of prior Copilot experience.

Finding 1. *Comprehension accuracy (i.e., the correctness of responses to program comprehension questions) was significantly higher for Real-time GazeCopilot-refactored code compared to code refactored by Standard Copilot Copilot and Pre-set GazeCopilot, with a moderate and a large effect size respectively. This suggests that integrating real-time gaze data allows prompts to more effectively address developers' comprehension challenges, which could in turn lead a more comprehensible refactored code.*

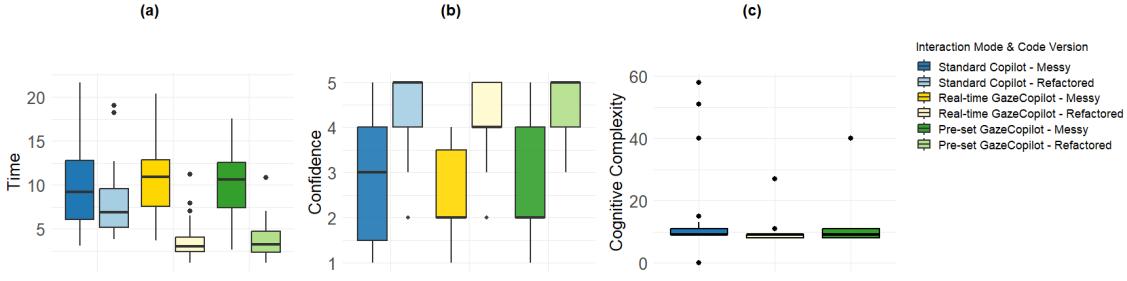

Fig. 5. Box plots for (a) comprehension time in minutes and (b) comprehension confidence by Interaction Mode and code snippet version (messy/refactored), and (c) refactored code snippets' cognitive complexity by Interaction Mode. Real-time GazeCopilot led to significantly faster comprehension (lower time), and the lowest mean cognitive complexity across Interaction Modes.

Comprehension Time. ART ANOVA results did not show a significant main effect of Interaction Mode, $F(2, 120) = 2.94$, $p = .057$, but showed a significant main effect of *version* (messy vs. refactored code), $F(1, 120) = 114.32$, $p < .001$, as well as a significant interaction (between Interaction Mode and *version*), $F(2, 120) = 9.47$, $p < .001$. Follow up post-hoc analysis revealed significant differences between two pairs only: comprehension time was significantly lower in Real-time GazeCopilot-refactored compared to Standard Copilot-refactored ($p = .0472$), and in Standard Copilot-refactored compared to Standard Copilot-messy ($p = .0102$).

Finding 2. *For refactored code, participants completed the task faster in Real-time GazeCopilot, with significantly shorter comprehension time than in Standard Copilot. This means that real-time gaze-informed prompts help developers process refactored code faster.*

Comprehension Confidence. ART ANOVA results indicated a significant main effect of version, $F(1, 130) = 301.80$, $p < .001$, with higher comprehension confidence for refactored code compared to messy code. We did not find significant effects for Interaction Mode or interaction.

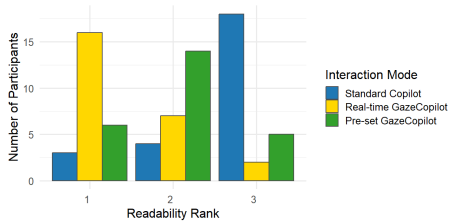

Fig. 6. Distribution of readability rankings for refactored code across Interaction Modes. The Real-time GazeCopilot Interaction Mode was most frequently ranked as most readable; the Standard Copilot was most often ranked as least readable.

Code Readability. Figure 6 shows that Real-time GazeCopilot-refactored code snippets were most often rated as the most readable (64%), whereas Standard Copilot was often ranked as least readable (72%). Pre-set GazeCopilot received intermediate rankings.

Descriptive statistics showed that the Real-time GazeCopilot Interaction Mode received the highest readability ($M = 1.44$, $Median = 1$, $SD = 0.707$), while Pre-set GazeCopilot received lower rankings ($M = 1.96$, $Median = 2$, $SD = 0.676$), and Standard Copilot Copilot had the lowest readability ($M = 2.60$, $Median = 3$, $SD = 0.707$).

The Friedman test indicated a statistically significant difference across Interaction Modes, $\chi^2(2) = 16.9, p = .000216$, with a moderate effect size (Kendall's $W = 0.338$).

Post-hoc comparisons indicated a statistically significant difference between Standard Copilot and Real-time GazeCopilot ($p_{adj} = .001$), with a large effect size ($r_{rb} = .7$). We did not find a significant difference between Standard Copilot and Pre-set GazeCopilot ($p_{adj} = .071$) or between Real-time GazeCopilot and Pre-set GazeCopilot ($p_{adj} = .106$).

> ***Finding 3***. *Participants ranked refactored code in Real-time GazeCopilot as more readable than in Standard Copilot with statistical significance and a large effect size. We did not find a significant difference between Standard Copilot and Pre-set GazeCopilot or between Real-time GazeCopilot and Pre-set GazeCopilot. This means that code refactored through real-time gaze-informed prompts is perceived by developers as more readable.*

Code Cognitive Complexity. Descriptive statistics show that the mean cognitive complexity was lowest (i.e., best) in code refactored in Real-time GazeCopilot ($M = 9.64$, $SD = 3.76$) , followed by Pre-set GazeCopilot ($M = 10.8$, $SD = 6.22$), and highest in Standard Copilot ($M = 14$, $SD = 14$).

> ***Finding 4***. *Refactored code in the Real-time GazeCopilot Interaction Mode had lower (i.e. better) mean cognitive complexity than in Standard Copilot and Pre-set GazeCopilot.*

Eye-Tracking Metrics. ART ANOVA showed significant main effects of code version (messy vs. refactored) on fixation duration ($F(1, 103.67) = 16.35, p < .001$), fixation count ($F(1, 104.21) = 26.37, p < .001$), and pupil dilation ($F(1, 107.44) = 22.62, p < .001$). The main effect of version on saccade length was marginally non-significant ($F(1, 103.43) = 3.28$, $p = 0.073$).

These results support our methodology of inferring code comprehension difficulties based on gaze data, as variations in these eye-tracking metrics reflect differences in cognitive processing demands across code versions (messy vs refactored).

> ***Finding 5***. *Code version (messy vs. refactored) had a significant effect on fixation duration, fixation count, and pupil dilation, with the messy version indicating higher cognitive demands. This supports our methodology, showing that gaze data reflects differences in cognitive processing and comprehension difficulties when comparing messy and refactored code.*

Impact of Real-time Gaze Signals on Refactoring Outcomes. Our results highlight the importance of tailoring AI-assisted refactoring to developers' real-time needs rather than relying on static prompts. The Real-time GazeCopilot approach enables selective refactoring by adapting its refactoring strategy to the developer's comprehension difficulties inferred from gaze behaviour. This minimizes redundant edits and reduces the effort required to revise code that the developer already understands.

In contrast, we applied the same hard-coded prompt for all participants in Pre-set GazeCopilot, assuming that all eye-tracking metrics exceeded thresholds and thus indicating maximum need for assistance. In Pre-set GazeCopilot, we found that Copilot's refactoring strategy was identical for 23 out of 25 participants, overgeneralizing refactoring and addressing more issues than the developer actually struggled with. Such excessive refactoring is not necessarily beneficial, as modifying code the developer already understands can introduce unnecessary changes and add cognitive and maintenance overhead without proportional gains [8, 41, 59, 64].

This is further supported by the analysis of comprehension and readability metrics, which shows that Real-time GazeCopilot-refactored code significantly outperformed Pre-set GazeCopilot-refactored code in comprehension accuracy, and was more frequently rated as more readable, as detailed in Section 5.2.

> ***Finding 6***. *Real-time GazeCopilot-refactored code significantly outperformed Pre-set GazeCopilot-refactored code in* ***comprehension accuracy*** *and was more frequently rated as more* ***readable****. This highlights that personalising refactoring to developers' real-time gaze behaviour (as done in Real-time GazeCopilot) is more effective than applying generic or overgeneralised improvements that do not rely on real-time gaze data (as done in Pre-set GazeCopilot), which in turn reduces unnecessary changes and the cognitive overhead required to review unneeded modifications.*

5.3 Programmers' Perception and Agency (RQ2)

Friedman test results did not indicate statistically significant differences between conditions for any of the user-perception questions.

Figure 7 shows the distribution of participants' answers. This suggests that the Real-time GazeCopilot prompt might be a promising addition to AI-coding assistants without negatively impacting key aspects of the developer's experience. One possible disadvantage of automatically generated prompts in the gaze-enhanced Interaction Modes could have been the loss of the sense of agency of the programmers. We found no statistically significant differences in perceived **control**, **control satisfaction**, **intuitiveness**, **ownership**, or **speed** across the three Interaction Modes.

This means there is no evidence that any of the Interaction Modes was perceived as more or less intuitive, controllable, or effective than the others.

> ***Finding 7.*** *We did not find evidence of differences in perceived* ***control***, ***control satisfaction***, ***intuitiveness***, ***ownership***, *or* ***speed*** *between the interaction modes. This suggest that Real-time GazeCopilot-prompts might be included to AI-coding assistants without negatively impacting key aspects of developers' experience, yet such integration needs further investigation.*

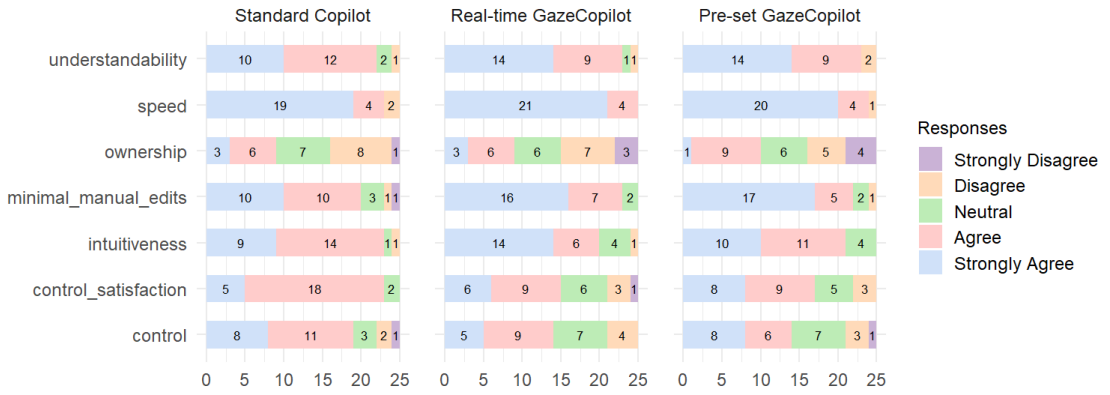

Fig. 7. Participants' responses to the survey questions on developers' experience and sense of agency. We did not find statistical differences between conditions for any of the questions.

5.4 Analysis of User Generated Prompts in Standard Copilot

To evaluate the quality of the prompts created by participants in the Standard Copilot treatment, we sought expert feedback from two experienced software developers with 7 and 10 years of professional programming experience, respectively. Both experts code daily in professional software development environments and have experience with AI coding assistants. Each expert was asked to rate the extent to which they found each participant-generated prompt meaningful for achieving the code refactoring task, using a 5-point Likert scale (1 = Strongly Disagree, 5 = Strongly Agree). The aggregated ratings ($M = 3.06$, $Median = 3$) suggest that, overall, experts perceived the prompts as neutral in quality.

These results indicate that the prompts generated by our participants are not meaningless and reflect the typical quality of prompts generated by developers, who often struggle to craft effective prompts, as noted in prior work [27, 45, 54, 65]. This further highlights the need for our real-time gaze-informed prompting approach, which significantly outperformed user-generated prompts in comprehension accuracy, comprehension time, and perceived code readability, as shown in Section 5.2. The user prompts and the expert ratings are publicly available in our replication package [4]).

6 Threats to Validity

We recognize potential threats to the validity of our findings and describe the steps we took to mitigate them throughout the study design and execution.

Internal validity. To minimize learning effects and ordering biases, we implemented a counter-balanced within-subjects design. The order of the three Interaction Modes (Standard Copilot, Real-time GazeCopilot, and Pre-set GazeCopilot) and the corresponding code snippets was counter-balanced across participants, resulting in six unique sequences (Table 2). This design choice helped mitigate potential confounding factors such as code-specific difficulty, task order effects, or participant fatigue. We also controlled for environmental factors by conducting all sessions in the same lab, using identical hardware, setup, and ambient lighting to ensure consistent eye-tracking accuracy.

External validity. To enhance generalizability and ensure a diverse and representative sample, the majority of participants (64%) had at least two years of programming experience, with (40%) reporting five or more years. Additionally, 40% coded on a weekly basis and 20% on a daily basis. This reflects a range of programming experience levels, enhancing the potential applicability of the study's results across the developer population.

To mitigate potential confounding factors, we employed three self-contained code snippets for our study tasks. While real-world code is typically more diverse, this controlled design minimized the influence of participants' familiarity with specific technologies, exposure to external dependencies, and fatigue, ensuring that observed effects could be attributed solely to the experimental conditions. Moreover, it allowed us to adhere to the study duration limits set by our institution's Ethics Board, further safeguarding participants' well-being.

Construct validity. We used established proxies for comprehension (e.g., PC questions), readability (e.g., cognitive complexity and rankings), and cognitive effort (e.g., gaze data). While we acknowledge that these proxies cannot fully capture the complexity of developers' cognitive processes, our use of validated measures and controlled conditions helps ensure that the interpretations remain as robust and meaningful as possible.

7 Discussion and Future Work

Integrating Real-time GazeCopilot into Development Environments. GazeCopilot provides a lightweight, IDE-integrated way to ease the cognitive load of manual prompt engineering, offering implicit support during code reading or refactoring. The system can be deployed in real-world applications as eye-tracking technology is advancing and becoming more accessible. Recent advances in visual computing allow accurate eye tracking using mobile front-facing cameras [44] and integrated webcams [79].

This growing availability of eye-tracking technology makes it feasible to directly integrate gaze-informed refactoring into real-world development environments. Real-time GazeCopilot can monitor developers' eye movements as they inspect complex or unclear code segments, automatically identifying areas that may cause cognitive strain or confusion. The system then generates targeted AI-assisted refactoring suggestions to improve readability without requiring explicit textual prompts. This capability allows developers to more quickly read, comprehend, and review code. Further development of GazeCopilot to include features such as explaining code sections based on developers' gaze behaviour can aid developers in debugging, refactoring, and onboarding processess. Future work should explore long-term use in real-world settings and scalability to large codebases and teams.

Benefits of Real-Time Gaze Data for Software Development. As LLMs have become integral to software practitioners' daily development activities, the workload for Quality Assurance (QA) activities (e.g., code review, bug triaging, testing) has increased significantly. According to a survey conducted with 500 software practitioners, 67% spend more

time debugging AI-generated code and 68% spend more time fixing AI-related security vulnerabilities [78]. Reducing the workload on QA activities requires detecting and fixing as many issues in the code base (which mostly contains human-written code) as possible at the earliest stages of the software development cycle. Improved code readability can facilitate better code comprehension, helping developers spot issues (e.g., bugs, security vulnerabilities) and fix them early. Monitoring developers' real-time gaze data can provide insights into which aspects of the code base need refactoring to improve code readability while avoiding excessive refactoring. Moreover, improving code readability benefits software maintainability. Due to developer turnover, it is essential that not only code owners but also other developers in organisations, including incoming software engineers, can understand the code base. Improving the code base readability also supports onboarding new software engineers.

Privacy and Agency. The integration of gaze-informed systems into software development workflows introduces critical considerations around privacy, transparency, and user agency. Prior work has shown that eye-tracking data, while valuable for modeling cognitive states and supporting adaptive interfaces, can inadvertently reveal sensitive information about users' intentions, emotional states, and even task content [3, 40]. These risks are amplified in professional programming contexts, where gaze traces may expose proprietary code or reveal aspects of developers' problem-solving strategies [6, 9]. As such, ensuring privacy-preserving and user-controllable deployment of gaze-informed AI assistants is essential.

To uphold developers' autonomy and trust, gaze-informed coding tools should prioritize local data processing and explicit user consent mechanisms. Local computation mitigates risks associated with transmitting sensitive gaze data to external servers, aligning with privacy-by-design principles in physiological computing [3, 40]. Moreover, user agency can be preserved by incorporating features such as session-level toggles—allowing developers to activate or deactivate gaze tracking at will—and transparent logging interfaces that clearly communicate when, how, and why gaze data are collected and processed [19, 69]. Such design affordances align with HCI guidelines emphasizing explainability and informed control in adaptive systems.

Balancing automation with respect for user autonomy thus requires that gaze-informed developer tools remain user-centered and consent-aware. Transparency in data handling and fine-grained control over sensing modalities are not only ethical imperatives but also practical enablers of user acceptance and long-term adoption. By embedding these safeguards, systems like Real-time GazeCopilot can leverage the cognitive benefits of gaze-based interaction while maintaining developers' privacy, control, and sense of ownership over their work.

Enabling Multi-modal AI Coding Assistants. This work repositions gaze from a passive observational tool into an active, interpretable signal for real-time AI adaptation. By leveraging gaze as a cognitive input channel, our study highlights the potential of multi-modal coding assistants that sense and respond to a developer's cognitive, behavioral, and affective states. Such systems could integrate complementary physiological signals, such as electroencephalography (EEG), galvanic skin response (GSR), or heart rate variability (HRV) to capture mental workload, stress, and engagement more comprehensively [16, 84]

Future adaptive AI-assisted development environments can achieve greater robustness and personalisation by incorporating multiple modalities. For instance, EEG and eye tracking have been used jointly to detect cognitive load and attention shifts in real time [21, 42], enabling interfaces that dynamically adapt to users' mental states. Similarly, multi-modal sensing has been shown to improve the detection of frustration and focus loss during complex problem-solving tasks [16, 38]. Translating these findings into software engineering and also referring to EEG and eye-tracking studies in software engineering [11, 30, 58], could help design and develop systems that modulate the level of AI

assistance, offering more explanations, detailed refactorings, or reduced intrusiveness—depending on the developer's current cognitive condition.

Towards Intent-Driven Software Development. Our work contributes to the emerging paradigm of intent-driven software development by advancing cognitively aware, adaptive tools that respond to developers' mental states and goals. SE 3.0 [35] envisions AI collaborators capable of understanding developer intent and adhering to software engineering principles, while vibe coding [15] emphasizes fluid, natural language interaction and real-time contextual awareness rather than rigid prompt structures. Building on these ideas, *GazeCopilot* extends the notion of intent inference by using gaze as an implicit signal of cognitive state, enabling more responsive and context-aware human–AI collaboration.

In this broader landscape of AI-assisted programming, the integration of multi-modal signals, such as gaze, EEG, and interaction logs offers new opportunities for intent recognition and adaptive collaboration strategies [14, 34]. Such signals can help an assistant discern when a developer is uncertain, fatigued, or deeply engaged, and dynamically tailor the timing, modality, or granularity of its support. This kind of adaptive responsiveness aligns with established principles of human–AI teaming, which emphasize fluid coordination, mutual awareness, and calibrated trust [20].

8 Conclusion

We introduce and evaluate Real-time GazeCopilot, a novel system that integrates real-time eye-tracking data into AI-assisted prompt design to enhance code readability and comprehension. By leveraging gaze metrics such as fixation duration and pupil dilation, Real-time GazeCopilot generates prompts that are aligned with the user's cognitive state. Our results show that real-time gaze-informed prompts significantly improve comprehension accuracy, comprehension time, and perceived code readability compared to standard text-based prompts. We demonstrate the potential of embedding implicit physiological signals to support more effective human-AI coding collaboration, especially for developers who struggle to express their intent.

References

[1] Naser Al Madi. 2022. 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*. 1–5.

[2] Salwa D Aljehane, Bonita Sharif, and Jonathan I Maletic. 2023. Studying developer eye movements to measure cognitive workload and visual effort for expertise assessment. *Proceedings of the ACM on Human-Computer Interaction* 7, ETRA (2023), 1–18.

[3] Nora Alsakar, Noura Alotaibi, Mohamed Khamis, and Simone Stumpf. 2025. Assessing and Mitigating the Privacy Implications of Eye Tracking on Handheld Mobile Devices. *ACM Transactions on Privacy and Security* (2025).

[4] Anonymous. [n. d.]. https://figshare.com/s/f08fec00e7b6e6badae1.

[5] Roman Bednarik and Markku Tukiainen. 2004. Visual attention tracking during program debugging. In *Proceedings of the third Nordic conference on Human-computer interaction*. 331–334.

[6] Raimundas Bednarik and Matti Tukiainen. 2004. Visual Attention Tracking during Program Debugging. In *Proceedings of the Third Nordic Conference on Human-Computer Interaction (NordiCHI 2004)*. ACM, 331–334. doi:10.1145/1028014.1028061

[7] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies* 18, 6 (1983), 543–554.

[8] Frank Buschmann. 2011. Gardening Your Architecture, Part 1: Refactoring . *IEEE Software* 28, 04 (July 2011), 92–94. doi:10.1109/MS.2011.76

[9] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *2015 IEEE 23rd International Conference on Program Comprehension*. 255–265. doi:10.1109/ICPC.2015.36

[10] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. 2011. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*. 1–9.

[11] John J. Camilleri, Alex R. E. Taylor, and Thomas Fritz. 2022. Psychophysiological Measures of Cognitive Load in Software Testing Tasks. *Human–Computer Interaction* 37, 5 (2022), 441–471. doi:10.1080/07370024.2021.1971815

[12] G Ann Campbell. 2018. Cognitive Complexity-A new way of measuring understandability. *SonarSource SA* 10 (2018).

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[14] Lucas Cibulski, Victor Ribeiro, Daniel Damian, and Igor Steinmacher. 2023. Cognitively Aware IDEs: Towards Human-Centered AI Support for Developers. *Empirical Software Engineering* 28, 5 (2023), 1–30. doi:10.1007/s10664-023-10360-4

[15] Wikipedia contributors. 2025. Vibe coding. https://en.wikipedia.org/wiki/Vibe_coding. Accessed: 2025-07-16.

[16] Benjamin Cowley, Perttu Hämäläinen, Kai Havukainen, and Katriina Heikura. 2016. Psychophysiological Methods for Adaptive Serious Games: A Review. *IEEE Transactions on Affective Computing* 7, 3 (2016), 273–288. doi:10.1109/TAFFC.2015.2446454

[17] Martha E Crosby and Jan Stelovsky. 1990. How do we read algorithms? A case study. *Computer* 23, 1 (1990), 25–35.

[18] Sarah D'Angelo and Darren Gergle. 2016. Gazed and confused: Understanding and designing shared gaze for remote collaboration. In *Proceedings of the 2016 chi conference on human factors in computing systems*. 2492–2496.

[19] Salvatore D'Angelo and Darren Gergle. 2016. Gazed and Confused: Understanding and Designing Shared Gaze for Remote Collaboration. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, 2492–2496. doi:10.1145/2858036.2858504

[20] Ewart J. de Visser, Robert W. Parasuraman, and Raja Parasuraman. 2020. Adaptive Human–Agent Teaming and the Dynamics of Trust. *Human Factors* 62, 3 (2020), 401–420. doi:10.1177/0018720819842311

[21] Sara Eivazi, Raimundas Bednarik, Bonita Sharif, Andrew Begel, Tobias Busjahn, and Carsten Schulte. 2017. An Eye-tracking and EEG Study on the Effect of Expertise in Debugging. In *Proceedings of the 2017 IEEE 25th International Conference on Program Comprehension (ICPC)*. IEEE, 146–156. doi:10.1109/ICPC.2017.25

[22] Ionut Daniel Fagadau, Leonardo Mariani, Daniela Micucci, and Oliviero Riganelli. 2024. Analyzing prompt influence on automated method generation: An empirical study with copilot. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 24–34.

[23] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. 2018. Empirical Software Engineering Experts on the Use of Students and Professionals in Experiments. *Empirical Softw. Engg.* 23, 1 (2018), 452–489.

[24] Xuefen Fang. 2001. Using a coding standard to improve program quality. In *Proceedings Second Asia-Pacific Conference on Quality Software*. 73–78. doi:10.1109/APAQS.2001.990004

[25] Franz Faul, Edgar Erdfelder, Axel Buchner, and Albert-Georg Lang. 2009. Statistical power analyses using G* Power 3.1: Tests for correlation and regression analyses. *Behavior research methods* 41, 4 (2009), 1149–1160.

[26] Janet Feigenspan, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2012. Measuring programming experience. In *2012 20th IEEE international conference on program comprehension (ICPC)*. IEEE, 73–82.

[27] Molly Q Feldman and Carolyn Jane Anderson. 2024. Non-expert programmers in the generative AI future. In *Proceedings of the 3rd Annual Meeting of the Symposium on Human-Computer Interaction for Work*. 1–19.

[28] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[29] Thomas Fritz, Andrew Begel, Sebastian C Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th international conference on software engineering*. 402–413.

[30] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using Psycho-Physiological Measures to Assess Task Difficulty in Software Development. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. ACM, New York, NY, USA, 402–413. doi:10.1145/2568225.2568266

[31] g00glen00b. 2017. SonarQube: Qualify cognitive complexity. https://stackoverflow.com/questions/45083653/sonarqube-qualify-cognitive-complexity/45084107#45084107. Accessed: 2025-10-27.

[32] Lisa Grabinger, Naser Al Madi, Roman Bednarik, Teresa Busjahn, Fabian Engl, Timur Ezer, Hans Gruber, Florian Hauser, Jonathan I Maletic, Unaizah Obaidellah, et al. 2025. A Cookbook for Eye Tracking in Software Engineering. In *Proceedings of the 6th European Conference on Software Engineering Education*. 60–76.

[33] Ebtesam Al Haque, Chris Brown, Thomas D LaToza, and Brittany Johnson. 2025. Towards Decoding Developer Cognition in the Age of AI Assistants. *arXiv preprint arXiv:2501.02684* (2025).

[34] Ahmed Hassan, Tim Menzies, Nicole Novielli, and Rahul Premraj. 2024. SE 3.0: Towards Intent-Driven and Cognitively Aware Software Engineering. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE, 1012–1024. doi:10.1109/ICSE48619.2024.00104

[35] Ahmed E Hassan, Gustavo A Oliva, Dayi Lin, Boyuan Chen, Zhen Ming, et al. 2024. Towards AI-native software engineering (SE 3.0): A vision and a challenge roadmap. *arXiv preprint arXiv:2410.06107* (2024).

[36] Mateusz Jaworski and Dariusz Piotrkowski. 2023. Study of software developers' experience using the Github Copilot Tool in the software development process. *arXiv preprint arXiv:2301.04991* (2023).

[37] John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. 2019. An empirical study assessing source code readability in comprehension. In *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 513–523.

[38] Imane Jraidi and Claude Frasson. 2013. Physiological and Emotional Adaptation in Intelligent Tutoring Systems. *Interactive Learning Environments* 21, 1 (2013), 19–33. doi:10.1080/10494820.2011.575791

[39] Zachary Karas, Aakash Bansal, Yifan Zhang, Toby Li, Collin McMillan, and Yu Huang. 2024. A tale of two comprehensions? analyzing student programmer attention during code summarization. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–37.

[40] Mohamed Khamis, Florian Alt, and Andreas Bulling. 2018. Privacy Implications of Eye Tracking: A Review and Future Outlook. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)* 1, 2 (2018), 1–36. doi:10.1145/3130979

[41] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649.

[42] Thomas Kosch, Andreas Hölzl, Kai Hecht, Albrecht Schmidt, and Mohamed Khamis. 2018. Your Eyes Tell: Leveraging Gaze and EEG for Cognitive Load Estimation. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, 1–13. doi:10.1145/3173574.3174003

[43] Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. 2024. Do large language models pay similar attention like human programmers when generating code? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2261–2284.

[44] Liqi Lai, Baohua Su, and Linwei She. 2025. Trends and Transformations: A Bibliometric Analysis of Eye-Tracking Research in Educational Technology. *Journal of Eye Movement Research* 18, 3 (2025), 23.

[45] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2024. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*. 1–13.

[46] Yu-Tzu Lin, Cheng-Chih Wu, Ting-Yun Hou, Yu-Chih Lin, Fang-Ying Yang, and Chia-Hu Chang. 2015. Tracking students' cognitive processes during program debugging—An eye-movement approach. *IEEE transactions on education* 59, 3 (2015), 175–186.

[47] Kevin Ljung and Javier Gonzalez-Huerta. 2022. "to clean code or not to clean code" a survey among practitioners. In *International Conference on Product-Focused Software Process Improvement*. Springer, 298–315.

[48] Alina Mailach, Dominik Gorgosch, Norbert Siegmund, and Janet Siegmund. 2025. "Ok Pal, we have to code that now": interaction patterns of programming beginners with a conversational chatbot. *Empirical Software Engineering* 30, 1 (2025), 34.

[49] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2024. Prompt Engineering in Large Language Models. In *Data Intelligence and Cognitive Informatics*, I. Jeena Jacob, Selwyn Piramuthu, and Przemyslaw Falkowski-Gilski (Eds.). Springer Nature Singapore, Singapore, 387–402.

[50] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[51] Suad Mohamed, Najma Ismail, Kimberly Amaya Hernandez, Abdullah Parvin, Michael Oliver, and Esteban Parra. 2025. Design of An Eye-Tracking Study Towards Assessing the Impact of Generative AI Use on Code Summarization. In *Proceedings of the 2025 Symposium on Eye Tracking Research and Applications*. 1–8.

[52] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–16.

[53] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 97, 13 pages. doi:10.1145/3597503.3639187

[54] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How beginning programmers and code llms (mis) read each other. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–26.

[55] Unaizah Obaidellah, Mohammed Al Haek, and Peter C-H Cheng. 2018. A survey on the usage of eye-tracking in computer programming. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–58.

[56] Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Vahid Meimand, Nan Zhang, Ben Ferrari-Church, and Satish Chandra. 2024. How much does AI impact development speed? An enterprise-based randomized controlled trial. *arXiv preprint arXiv:2410.12944* (2024).

[57] Ki Park, J. Johnson, C. S. Peterson, et al. 2024. An Eye Tracking Study Assessing Source Code Readability Rules for Program Comprehension. *Empirical Software Engineering* 29, 160 (2024). doi:10.1007/s10664-024-10532-x

[58] Norman Peitek, David Winter, Janet Siegmund, Bonita Sharif, and Sven Apel. 2022. Eye Movements and Brain Activity during Source Code Reading: Replication and Extension. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. ACM, New York, NY, USA, 593–605. doi:10.1145/3540250.3549150

[59] Victor Hugo Santiago C Pinto, Alberto Luiz Oliveira Tavares de Souza, Yuri Matheus Barboza de Oliveira, and Danilo Monteiro Ribeiro. 2021. Cognitive-Driven Development: Preliminary Results on Software Refactorings.. In *ENASE*. 92–102.

[60] Cristiano Politowski, Foutse Khomh, Simone Romano, Giuseppe Scanniello, Fabio Petrillo, Yann-Gaël Guéhéneuc, and Abdou Maiga. 2020. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. *Information and Software Technology* 122 (2020), 106278.

[61] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The widening gap: The benefits and harms of generative ai for novice programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1*. 469–486.

[62] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*. 390–401.

[63] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 491–514.

[64] Giulia Sellitto, Emanuele Iannone, Zadia Codabux, Valentina Lenarduzzi, Andrea De Lucia, Fabio Palomba, and Filomena Ferrucci. 2022. Toward understanding the impact of refactoring on program comprehension. In *2022 IEEE international conference on software analysis, evolution and*

reengineering (SANER). IEEE, 731–742.

[65] Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. 2025. Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Information and Software Technology* 178 (2025), 107610.

[66] Anshul Shah, Anya Chernova, Elena Tomson, Leo Porter, William G Griswold, and Adalbert Gerald Soosai Raj. 2025. Students' Use of GitHub Copilot for Working with Large Code Bases. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*. 1050–1056.

[67] Zohreh Sharafi, Timothy Shaffer, Bonita Sharif, and Yann-Gaël Guéhéneuc. 2015. Eye-tracking metrics in software engineering. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 96–103.

[68] Vijay Kartik Sikha, Dayakar Siramgari, and Laxminarayana Korada. 2023. Mastering Prompt Engineering: Optimizing Interaction with Generative AI Agents. *Journal of Engineering and Applied Sciences Technology. SRC/JEAST-E117. DOI: doi. org/10.47363/JEAST/2023 (5) E117 J Eng App Sci Technol* 5, 6 (2023), 2–8.

[69] Simone Stumpf, Mohan Kankanhalli, Aditi Paul, and Peter Brusilovsky. 2020. Explanations Considered Harmful? User Agency in AI-Assisted Systems. *ACM Transactions on Computer–Human Interaction (TOCHI)* 27, 6 (2020), 1–40. doi:10.1145/3411764

[70] Ningzhi Tang, Junwen An, Meng Chen, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2024. 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*. 119–123.

[71] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2024. Developer behaviors in validating and repairing llm-generated code using ide and eye tracking. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 40–46.

[72] Yahya Tashtoush, Zeinab Odat, Izzat M Alsmadi, and Maryan Yatim. 2013. Impact of programming features on code readability. (2013).

[73] Tenet. 2024. GitHub Copilot Usage Data and Statistics. https://www.wearetenet.com/blog/github-copilot-usage-data-statistics Accessed: 2025-07-03.

[74] Tobii AB. 2025. Tobii Pro SDK. https://www.tobii.com/products/software/applications-and-developer-kits/tobii-pro-sdk. Accessed: 2025-07-03.

[75] Tobii AB. 2025. Tobii Pro Spark. https://www.tobii.com/products/eye-trackers/screen-based/tobii-pro-spark. Accessed: 2025-07-03.

[76] Adam Tornhill and Markus Borg. 2022. Code red: the business impact of code quality-a quantitative study of 39 proprietary production codebases. In *Proceedings of the International Conference on Technical Debt*. 11–20.

[77] Maureen Villamor and Ma Mercedes Rodrigo. 2018. Predicting successful collaboration in a pair programming eye tracking experiment. In *Adjunct Publication of the 26th Conference on User Modeling, Adaptation and Personalization*. 263–268.

[78] Mike Vizard. 2025. Survey: AI Tools are Increasing Amount of Bad Code Needing to be Fixed. https://devops.com/survey-ai-tools-are-increasing-amount-of-bad-code-needing-to-be-fixed/. Accessed: 2025-11-01.

[79] Katarzyna Wisiecka, Krzysztof Krejtz, Izabela Krejtz, Damian Sromek, Adam Cellary, Beata Lewandowska, and Andrew Duchowski. 2022. Comparison of webcam and remote eye tracking. In *2022 Symposium on eye tracking research and applications*. 1–7.

[80] Andreas Wulff-Jensen, Kevin Ruder, Evangelia Triantafyllou, and Luis Emilio Bruni. 2019. Gaze Strategies Can Reveal the Impact of Source Code Features on the Cognitive Load of Novice Programmers. In *Advances in Neuroergonomics and Cognitive Engineering*, Hasan Ayaz and Lukasz Mazur (Eds.). Springer International Publishing, Cham, 91–100.

[81] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.

[82] Wudao Yang and Unaizah Obaidellah. 2024. Attention Dynamics in Programming: Eye Gaze Patterns of High-vs. Low-Ability Novice Coders. In *Proceedings of the 2024 Symposium on Eye Tracking Research and Applications*. 1–6.

[83] Thomas Y Yeh, Karena Tran, Ge Gao, Tyler Yu, Wai On Fong, and Tzu-Yi Chen. 2025. Bridging Novice Programmers and LLMs with Interactivity. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*. 1295–1301.

[84] Thorsten O. Zander and Christian Kothe. 2011. Towards Passive Brain–Computer Interfaces: Applying Brain–Computer Interface Technology to Human–Machine Systems in General. *Journal of Neural Engineering* 8, 2 (2011), 025005. doi:10.1088/1741-2560/8/2/025005

[85] Xuan Zhang and I Scott MacKenzie. 2007. Evaluating eye tracking with ISO 9241-Part 9. In *Human-Computer Interaction. HCI Intelligent Multimodal Interaction Environments: 12th International Conference, HCI International 2007, Beijing, China, July 22-27, 2007, Proceedings, Part III 12*. Springer, 779–788.

[86] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2024. Measuring GitHub Copilot's Impact on Productivity. *Commun. ACM* 67, 3 (March 2024), 54–63. doi:10.1145/3633453