RESEARCH-ARTICLE

The Impact of Generative AI Coding Assistants on Developers Who Are Visually Impaired

CLAUDIA FLORES-SAVIAGA, Northeastern University, Boston, MA, United States

BENJAMIN V HANRAHAN, Microsoft Corporation, Redmond, WA, United States

KASHIF IMTEYAZ, Northeastern University, Boston, MA, United States

STEVEN CLARKE, Microsoft Corporation, Redmond, WA, United States

SAIPH SAVAGE, National Autonomous University of Mexico, Mexico City, Mexico

PDF Download
3706598.3714008.pdf
05 January 2026
Total Citations: 3
Total Downloads: 2141

Published: 26 April 2025

Citation in BibTeX format

CHI 2025: CHI Conference on Human Factors in Computing Systems
April 26 - May 1, 2025
Yokohama, Japan

Conference Sponsors:
SIGCHI

The Impact of Generative AI Coding Assistants on Developers Who Are Visually Impaired

Claudia Flores-Saviaga
Northeastern University*
Boston, Massachusetts, USA
floressaviaga.c@northeastern.edu

Benjamin V. Hanrahan
Microsoft
Redmond, Washington, USA
benhanrahan@microsoft.com

Kashif Imteyaz
Northeastern University
Boston, Massachusetts, USA
imteyaz.k@northeastern.edu

Steven Clarke
Microsoft
Edinburgh, United Kingdom
stevencl@microsoft.com

Saiph Savage*
Universidad Nacional Autonoma de Mexico (UNAM)
Mexico City, Mexico
s.savage@northeastern.edu

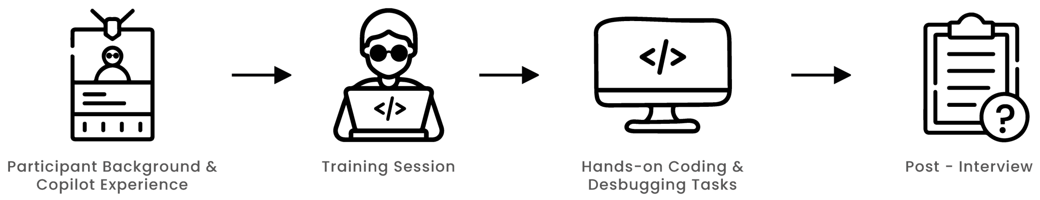

Participant Background & Copilot Experience
Training Session
Hands-on Coding & Desbugging Tasks
Post - Interview

Figure 1: Overview of our research to analyze how developers who are visually impaired interact with AI coding assistants.

Abstract

The rapid adoption of generative AI in software development has impacted the industry, yet its effects on developers with visual impairments remain largely unexplored. To address this gap, we used an *Activity Theory* framework to examine how developers with visual impairments interact with AI coding assistants. For this purpose, we conducted a study where developers who are visually impaired completed a series of programming tasks using a generative AI coding assistant. We uncovered that, while participants found the AI assistant beneficial and reported significant advantages, they also highlighted accessibility challenges. Specifically, the AI coding assistant often exacerbated existing accessibility barriers and introduced new challenges. For example, it overwhelmed users with an excessive number of suggestions, leading developers who are visually impaired to express a desire for "AI timeouts." Additionally, the generative AI coding assistant made it more difficult for developers to switch contexts between the AI-generated content and their own code. Despite these challenges, participants were optimistic about the potential of AI coding assistants to transform the coding experience for developers with visual impairments. Our findings emphasize the need to apply *activity-centered design* principles to generative AI assistants, ensuring they better align with user behaviors and address specific accessibility needs. This approach can enable the assistants to provide more intuitive, inclusive, and effective experiences, while also contributing to the broader goal of enhancing accessibility in software development.

CCS Concepts

• **Human-centered computing → Empirical studies in interaction design**; **Empirical studies in accessibility**.

Keywords

generative ai, accessibility, ai coding assistants, assistive technology

ACM Reference Format:
Claudia Flores-Saviaga, Benjamin V. Hanrahan, Kashif Imteyaz, Steven Clarke, and Saiph Savage*. 2025. The Impact of Generative AI Coding Assistants on Developers Who Are Visually Impaired. In *CHI Conference on Human Factors in Computing Systems (CHI '25), April 26–May 01, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3706598.3714008

1 Introduction

The integration of generative artificial intelligence (AI) into software development is fundamentally transforming coding practices [55]. AI coding assistants, such as GitHub Copilot [34], provide functionalities like auto-completion, code generation, and test generation [76], which have the potential to revolutionize how developers work [73]. Amidst this surge of new capabilities driven by AI, an important question emerges: How do these generative AI coding assistants impact developers with visual impairments?

The convergence of AI and accessibility in coding presents both tremendous opportunities and intricate challenges [55, 67]. On one hand, AI has the potential to make coding more accessible for developers with visual impairments by reducing manual coding tasks and providing intelligent assistance [89]. On the other hand, if these AI coding assistants are not designed with accessibility in mind, they may inadvertently create new barriers or worsen existing ones [67, 103]. This risk is especially high if the AI coding assistants depend heavily on visual cues or employ interaction methods incompatible with screen readers [61, 86, 109].

In this context, it is important to understand that the relationship between accessibility and usability in AI assistants mirrors long-standing challenges in web accessibility [38, 51], particularly for visually impaired users [14, 43]. Similar to rich internet applications [9, 72], AI assistants often introduce dynamic content and complex interactions that traditional accessibility approaches may not fully address [39, 47]. For instance, Petrie and Kheir [74] found that existing accessibility guidelines fail to capture many of the usability problems encountered by visually impaired users when interacting with dynamic content. This gap could be especially exacerbated in AI coding assistants [2, 65, 71], where AI-generated suggestions, real-time code generation, and sophisticated user interfaces can present new hurdles [39, 41]. Just as screen readers struggle with dynamic web content, AJAX updates, and automatic refreshes [20], developers who are visually impaired may experience similar difficulties when interacting with rapidly changing AI-generated code suggestions and interface elements [54]. Consequently, simply following basic accessibility rules might not be enough to make AI coding assistants truly easy to use. We might need to rethink how to make these assistants more accessible for everyone. The challenge is that we do not fully understand the difficulties and benefits that visually impaired developers experience when using AI coding assistants. Closing this knowledge gap is important—not just for accessibility, but also for making AI-assisted coding tools more user-friendly and effective for all developers, including those with different levels of vision.

Our study addresses this research gap by investigating what unique challenges and opportunities AI-assisted coding tools present for developers with visual impairments. Our research is driven by the following research question:

- RQ1: What challenges do AI-assisted coding tools pose for developers who are visually impaired, and what opportunities do they offer to empower and enhance their work?

To study this question, we use the Activity Theory framework [44], which helps analyze how tools influence human activity and identifies *contradictions* that occur when a tool's design does not fit well with users' workflows and needs [12]. We applied this framework to examine a qualitative study we conducted with 10 visually impaired developers of varying experience levels. These developers used an AI coding assistant, GitHub Copilot, to complete a coding task. During the study, we observed their real-time interactions with the AI assistant, noting both challenges and opportunities. After the task, we conducted interviews to gain deeper insights into their experiences and the difficulties they faced while using the AI coding assistant.

Our findings reveal a complex landscape where AI coding assistants can both improve coding efficiency and introduce new accessibility challenges. Based on our findings, we outline a roadmap for the next generation of accessible AI coding assistants. By analyzing the experiences of visually impaired developers using AI-assisted coding tools, we derive key design insights that can reshape how accessibility is approached in AI-driven software development.

The contributions of this paper are twofold:

- We conduct a comprehensive analysis of the accessibility challenges and benefits of AI coding assistants, using real-world insights from developers who are visually impaired.
- We propose a set of design recommendations to make AI coding assistants more accessible and inclusive for developers with visual impairments.

2 Related Work

2.1 Accessibility and Tools for Developers who are Visually Impaired

Accessibility in general has been a subject of ongoing research [17, 35, 36, 85]. Several studies have shed light on the significant challenges developers who are visually impaired face with coding and the techniques these programmers employ to overcome these challenges. Mountapmbeme et al. [63] categorized these challenges into five main areas: code navigation, code comprehension, code editing, code debugging, and code skimming. Albusays and Ludi [5] found persistent challenges in code navigation, accessing diagrams, debugging, and UI layout. Their study highlighted a strong preference for text editors over IDEs due to accessibility issues, reduced complexity, and better compatibility with assistive technologies.

Further studies by Mealin and Murphy-Hill [59] and Baker et al. [11] explored specific tools and techniques that developers who are visually impaired use. Mealin and Murphy-Hill found that many developers who are visually impaired rely on text editors rather than IDEs due to accessibility issues employing unique practices like "out-of-context editing," where blocks of code are copied, edited separately, and pasted back. This preference for text editors has been corroborated by other researchers [6, 63], who attribute it to specific challenges such as navigating line-by-line, understanding indentation, and managing nested code structures [56, 99]. To address this issue, Baker et al. [11] created StructJumper, a plugin that generates a hierarchical tree structure of code for easier navigation. Debugging, presents another significant challenge for developers who are visually impaired, largely due to the reliance on visual interfaces in debugging tools, which screen readers struggle to interpret effectively [5, 78]. These challenges have led the development of tools like Wicked Audio Debugger (WAD) [94] , a tool that provides audio descriptions of programs during execution; CodeTalk [78], a Visual Studio plugin that introduces "TalkPoints" for audio-based debugging; and CodeWalk, a tool by Potluri et al. [77], which facilitates accessible, remote, synchronous code review and refactoring activities by tethering collaborators' cursors with the host of a Live Share session. Additionally, Haque et al. [27] introduced Grid-Coding, a paradigm designed to improve the accessibility of coding environments by representing code in a structured 2D grid, allowing developers who are visually impaired to navigate and edit code more effectively.

Conversational interfaces have also shown promise [15], Ludi et al. [57], found that speech-based cues generally provided the best performance for comprehension and navigation tasks. Phutane et al. [75] explored the use of voice commands to reduce cognitive load. Meanwhile, Stefik et al. [95] created Sodbeans, an IDE that relies on audio cues for debugging, while Smith et al. [90] developed a tool to allow developers to navigate the tree structure of files in Eclipse.

While previous research has significantly advanced accessibility for developers who are visually impaired, these findings may not fully apply to the new landscape shaped by generative AI-assisted coding tools such as GitHub Copilot. The introduction of these tools brings new challenges and opportunities in accessible programming, an area that remains largely unexamined. The majority of current studies were conducted before the emergence of generative AI in coding environments, resulting in a critical knowledge gap regarding the impact and potential of these advanced tools for developers who are visually impaired.

2.2 AI-assisted Coding Environments

The integration of AI into software development tools has significantly impacted coding practices, with AI-assisted coding assistants like GitHub Copilot showing promise in improving developer productivity. Ziegler et al. [108] found that Copilot increases users' feelings of productivity, with almost a third of its proposed code completions being accepted. In a controlled experiment, Peng et al.[73] demonstrated that software developers using Github Copilot were able to complete programming tasks significantly faster than those without such assistance. However, Vaithilingam et al. [100] noted that while most participants preferred using Copilot in daily programming tasks, they often faced difficulties in understanding, editing, and debugging code snippets generated by Copilot, which significantly hindered their task-solving effectiveness. This was confirmed by Dakhel et al [26], who noted that the effectiveness of tools like Github Copilot depends on the developer's level of expertise, as less experienced developers often lack the necessary skills to effectively evaluate AI-generated code suggestions.

A recent survey of developers by Liang et al. [55] revealed that the primary motivations for using AI programming assistants include reducing keystrokes, finishing programming tasks quickly, and recalling syntax. Developers reported that a median of 30.5% of their code was written with help from tools like Copilot.

Studies like those by Barke et al.[66] and Wu et al. [58] provide insight into the varying modes of interaction with AI coding assistants. Barke et al. [66] identified "acceleration mode" and "exploration mode" as two broad categories of Copilot use, while Wu et al. [58] compared human-human pair programming with human-AI pair programming, highlighting differences in collaboration dynamics. However, there is still a significant gap in understanding how these AI tools impact developers with accessibility needs.

2.3 Activity Theory

Activity Theory is a type of "conceptual framework" [80, 82, 88, 91]. A conceptual framework is an analytical tool or structure used to organize and guide research, projects, or problem-solving efforts [40, 60]. The foundational concept of Activity Theory is the "activity", a series of goal directed actions [53] that aim to achieve specific objectives [52]. These actions are mediated by artifacts, the instruments through which individuals interact with their objectives. Actions themselves are performed via routinized operations, in which individuals are not conscious of or focused on these operations. This process is illustrated in Figure 2. For example, developers

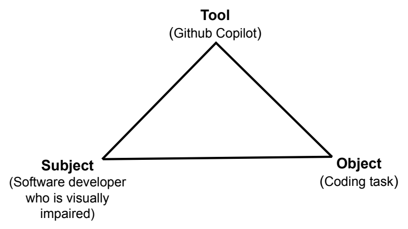
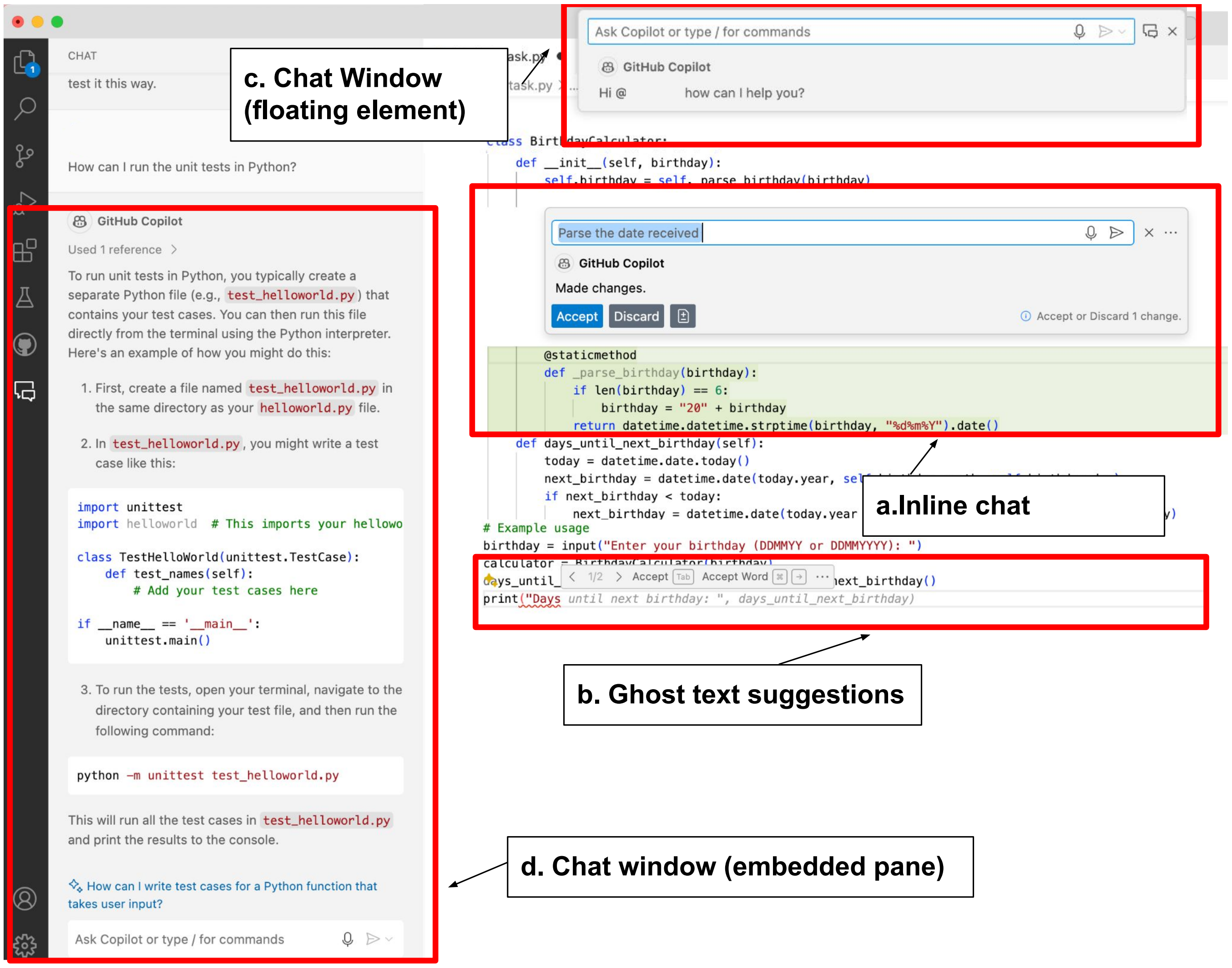


Figure 2: Diagram showing the relationship between developers who are visually impaired (subject), coding tasks (object), and GitHub Copilot (tool) in the Activity Theory framework.

who are visually impaired (the subjects) may have specific goals (objects) related to completing programming tasks. In this context, an AI programming tool acts as the mediating artifact (tool).

Bødker proposed Activity Theory as a foundational framework for HCI [18], emphasizing its potential to guide the design of interactive systems by focusing on the dynamic interplay between users, their goals, and the tools users employ (interactive systems) [19]. Bødker emphasized that tools should evolve over time to meet users' needs and adapt to the activities they mediate[18], highlighting the importance of designing systems that adjust to changing workflows and contexts to remain effective[19]. Within Bødker's definition of Activity Theory for HCI [18], *contradictions* are discrepancies or tensions that arise between the tools, the goals, and the users, that must be addressed [28]. These *contradictions* are not obstacles but drivers of change, pointing out areas where tools or processes need to evolve to better meet user needs [29]. Within this space, Activity Theory also introduces the concept of *misalignments*, which are localized issues or practical mismatches. Unlike *contradictions*, *misalignments* hinder usability and effectiveness for end-users but may not necessitate systemic changes.

In this paper, we use Activity Theory as a framework to examine the design of AI-assisted coding tools for developers who are visually impaired. We chose this approach because Activity Theory has been widely used for decades to study tools and technologies designed for diverse groups, including individuals with disabilities [12, 18, 28, 44, 45, 81, 97]. For example, Baldwin et al. [12] applied Activity Theory to investigate the design of tactile devices and enhanced auditory tools, examining whether these technologies align with the unique workflows of users who have no or low-vision. Similarly, Szymczak [97] applied Activity Theory to analyze how audio-haptic technologies mediated interactions and supported the goals of individuals who are visually impaired, focusing on how these technologies could assist users in interacting with 2D representations, such as maps and drawings. Tlili et al. [98], also applied Activity Theory to review the use of game-based learning for learners with disabilities and identify inconsistencies in stakeholder involvement, variability in the use of educational technology, and difficulties in standardizing performance measures. Robins [81] applied Activity Theory to analyze and design accessibility in video games, focusing on the interaction between visually impaired players, game goals, and mediating tools like audio-haptic technologies and game mechanics.


c. Chat Window (floating element)
a.Inline chat
b. Ghost text suggestions
d. Chat window (embedded pane)
CHAT
task.py
GitHub Copilot
Used 1 reference
Accept
Discard
Accept Tab
Accept Word
1/2
Ask Copilot or type / for commands

Ask Copilot or type / for commands

GitHub Copilot

Hi @ how can I help you?

test it this way.

How can I run the unit tests in Python?

GitHub Copilot

Used 1 reference

To run unit tests in Python, you typically create a separate Python file (e.g., `test_helloworld.py`) that contains your test cases. You can then run this file directly from the terminal using the Python interpreter. Here's an example of how you might do this:

1. First, create a file named `test_helloworld.py` in the same directory as your `helloworld.py` file.
2. In `test_helloworld.py`, you might write a test case like this:

```
import unittest
import helloworld  # This imports your hellowo

class TestHelloWorld(unittest.TestCase):
    def test_names(self):
        # Add your test cases here

if __name__ == '__main__':
    unittest.main()
```

3. To run the tests, open your terminal, navigate to the directory containing your test file, and then run the following command:

```
python -m unittest test_helloworld.py
```

This will run all the test cases in `test_helloworld.py` and print the results to the console.

How can I write test cases for a Python function that takes user input?

Ask Copilot or type / for commands

Parse the date received

GitHub Copilot

Made changes.

Accept or Discard 1 change.

```
class BirthdayCalculator:
    def __init__(self, birthday):
        self.birthday = self._parse_birthday(birthday)

        @staticmethod
        def _parse_birthday(birthday):
            if len(birthday) == 6:
                birthday = "20" + birthday
            return datetime.datetime.strptime(birthday, "%d%m%Y").date()
    def days_until_next_birthday(self):
        today = datetime.date.today()
        next_birthday = datetime.date(today.year, sel
        if next_birthday < today:
            next_birthday = datetime.date(today.year
# Example usage
birthday = input("Enter your birthday (DDMMYY or DDMMYYYY): ")
calculator = BirthdayCalculator(birthday)
days_until_                                  next_birthday()
print("Days until next birthday: ", days_until_next_birthday)
```

Figure 3: GitHub Copilot interaction interfaces. (a) Inline chat window for quick command input and AI engagement. (b) Ghost text suggestions dynamically generated as the user types. (c) Floating chat window for temporary interactions, ideal for quick-access queries. (d) Embedded pane chat window for ongoing, more extensive conversations with AI, allowing for sustained reference and deeper coding assistance.

Building on this foundation, our paper applies Activity Theory to examine how AI-assisted coding tools, particularly GitHub Copilot, mediate and influence the tasks of developers who are visually impaired. Activity Theory provides a particularly useful analysis framework since the introduction of AI coding assistants may operationalize some actions that have not yet become routine for developers. Additionally, we use Activity Theory to guide design recommendations aimed at improving these tools for this population. To achieve this, we analyze both *contradictions*—systemic tensions within the activity system—and *misalignments*—localized issues that disrupt usability. By addressing these challenges, we ensure AI tools meet developers' needs.

2.4 Copilot

The GitHub Copilot interface provides multiple ways for developers to interact with its AI-driven coding assistant. One primary method is through its **inline chat** (Fig. 3a), where users can type commands or questions directly within the code editor. This allows for seamless, quick interactions without disrupting the coding workflow.

Additionally, developers can engage with Copilot using the **chat window**, which can appear either as a *floating element* (Fig. 3c) or as an *embedded pane* within the interface (Fig. 3d). An embedded pane refers to a UI component that is fixed within the main window, allowing for continuous visibility and interaction without obstructing other elements on the screen. The floating element is useful for quick, temporary queries, while the embedded pane is

P#	Gender	Age	Exp.(yrs)	Vision Status	AI-Coding Tool Experience	Coding Proficiency	Pref. Lang. For Study	Task Completed
P1	Male	32	8	No vision	GitHub Copilot	C#, JavaScript, Python, Ruby	Python	Yes
P2	Male	34	12	Low vision	GitHub Copilot	PHP, C#, JavaScript	C#	Yes
P3	Male	38	9	Low vision	CodeWhisperer, Codellama, ChatGPT	C++, Python	Python	Yes
P4	Male	43	22	Low vision	ChatGPT	JavaScript, TypeScript, C#	C#	Yes
P5	Male	26	6	No vision	GitHub Copilot, ChatGPT	Python	Python	Yes
P6	Male	36	17	Low vision	GitHub Copilot	PHP, Ruby, Swift, Go, Python, JavaScript, Kotlin	Python	Yes
P7	Male	58	23	No vision	GitHub Copilot	Python, SQL, PowerShell	Python	Yes
P8	Male	54	32	No vision	ChatGPT	C, R, Ruby, TypeScript, Swift, Python, Kotlin	Python	Yes
P9	Male	42	4	No vision	ChatGPT	JavaScript, PHP, Python, HTML, CSS	Python	Yes
P10	Male	24	2	No vision	Claude, ChatGPT, Gemini	C#, HTML, PHP	C#	Yes

Table 1: Participant Demographics, Vision Status, Experience with AI Coding Assistants, and Programming Background.

better suited for extended interactions where users may need to frequently reference past exchanges. This flexibility enables developers to choose an interaction style that best fits their workflow and preferences.

Another key interaction method is through **ghost text suggestions** (Fig. 3b). As the developer types, Copilot continuously analyzes the context and generates relevant code suggestions in real-time. These suggestions appear as semi-transparent text within the editor, allowing developers to seamlessly integrate them into their code. Developers can choose to accept, modify, or ignore these suggestions based on their needs. Beyond real-time suggestions, developers can also request code completions by writing natural language comments that describe the desired functionality. Copilot then processes these descriptions and generates corresponding code snippets, helping developers implement features more efficiently.

It is important to note that GitHub Copilot is not a standalone code editor but an extension designed to work within Visual Studio Code. The chat windows, including both the floating and embedded pane versions, are features introduced by the GitHub Copilot extension and are not part of Visual Studio Code by default. These chat interfaces allow users to interact directly with AI-generated coding suggestions within their coding environment.

GitHub Copilot was selected as the primary AI coding assistant for this study because, at the time of research, it was the most widely used tool of its kind [33, 92]. We chose Visual Studio Code as the development environment due to its popularity and accessibility, as well as its seamless integration with GitHub Copilot [93, 101]. This setup enabled us to explore how these tools influence the coding experience of developers who are visually impaired.

2.5 Research Gap

Despite progress in accessibility for developers who are visually impaired [13, 23, 65], generative AI coding assistants like GitHub Copilot introduce new challenges and opportunities that remain under-explored. Most existing research focuses on traditional coding tools [1, 49], overlooking complexities unique to AI-assisted coding. Our study examines these accessibility issues to ensure AI assistants like GitHub Copilot are inclusive and support equal access for visually impaired developers.

3 Methods

To study the relationship between AI coding assistants and accessibility, we conducted a study with developers who are visually impaired. Our goal was to understand their experiences, challenges, and strategies when using an AI coding assistant, specifically GitHub Copilot. Using Activity Theory as our framework, we analyzed how Copilot (tool) mediates the interaction between visually impaired developers (participants) and their coding tasks (object). Our analysis highlights the contradictions and misalignments that occur when Copilot's design falls short of meeting the unique needs of developers who are visually impaired, while also uncovering new opportunities that emerge in this environment.

3.1 Participant Recruitment

We recruited 10 software developers who were visually impaired, including 4 with low vision and 6 with no vision, with experience ranging from 2 to 32 years in the field (see Table 1). Participants were sourced through professional networks, accessibility-focused online forums, and organizations supporting professionals in tech with visual impairments. To participate, developers needed some familiarity with AI coding assistants, though extensive experience with Copilot was not required. All participants identified as male. Given the specialized nature of developers who are visually impaired, our recruitment pool was naturally limited. However, our sample size aligns with prior HCI studies on underrepresented populations [25, 30, 69, 79, 83]. Despite these constraints, our study provides valuable insights into the accessibility challenges and opportunities for this underrepresented group. Each participant received $50 USD (or its equivalent in local currency) as compensation for their time.

3.2 Study setup

We conducted our study remotely, ensuring that participants could use their preferred accessible setup. Each session lasted approximately 90 minutes and followed four sequential phases:

(1) **Participant Background and Copilot Experience:** We asked participants to describe their professional and technical backgrounds and share their prior experience with AI coding assistants like GitHub Copilot.
(2) **Training Session:** We provided participants with a training session that introduced GitHub Copilot and its key features.

The session also covered how Copilot integrates with accessible development environments and assistive technologies participants were already using. This ensured that all participants had a solid understanding of how to interact with Copilot before starting the coding tasks.

(3) **Hands-on Coding and Debugging Tasks:** We instructed participants to complete a coding task followed by a debugging task using GitHub Copilot in Visual Studio Code.
   - **Programming Task:** Following prior work [46], we asked participants to write a program that calculates the number of days until the user's next birthday. The program required a birthday string input in either `DDMMYY` or `DDMMYYYY` format and a class implementation to handle date calculations. We also asked participants to write unit tests for their class. Throughout the task, we encouraged them to use the "think-aloud" protocol, verbalizing their thought process while interacting with Copilot.
     We selected this task because:
     - It incorporates common programming concepts such as string manipulation, date calculations, and object-oriented principles, making it relevant to participants' typical coding workflows.
     - It does not require specialized knowledge of specific programming frameworks, simplifying participant recruitment.
   - **Debugging Task:** We then asked participants to engage in a debugging session, where they identified and fixed errors in the code generated by Copilot. During this session, we prompted them to discuss:
     - Their usual approach to evaluating code accuracy.
     - The methods they used to handle encountered issues.
     - Whether Copilot simplified or complicated their debugging process.

(4) **Post-Interview:** After completing the programming and debugging tasks, we conducted a semi-structured interview with each participant. We asked them to explain their approach to organizing their code and describe any adjustments they made based on Copilot's suggestions. They also shared the challenges they encountered and the solutions they implemented. Additionally, we encouraged them to reflect on how Copilot influenced the complexity of their tasks, providing examples of when it was particularly helpful or challenging. Our Appendix contains further details on the post-study interview questions.

A diagram illustrating our study setup is shown in Figure 1.

3.3 Data Collection and Analysis

We recorded and transcribed all interviews, capturing both verbal responses and relevant audio cues from participants' screen readers and other assistive technologies. We analyzed the data using thematic coding, focusing on significant events that participants experienced while using GitHub Copilot. Two of us independently coded the transcripts, identifying emerging themes and patterns. We paid special attention to pivotal moments that either enhanced or hindered participants' coding processes. These critical incidents provided concrete examples of how Copilot mediated accessibility, revealing specific ways in which the tool aligned with or diverged from the non-visual coding strategies used by developers who are visually impaired. To deepen our analysis, we integrated *Activity Theory*, which allowed us to examine the opportunities and contradictions that visually impaired developers encountered when interacting with an AI coding assistant. In particular, Activity Theory helped us explore within our themes:

(1) **Activity-Centric Lens:** how developers who are visually impaired (*subjects*) used GitHub Copilot (*tool*) to complete their coding tasks (*object*). By applying Activity Theory, we mapped the observed dynamics within the activity system, identifying key interactions and tensions among its components. This framework helped us understand how Copilot influenced developers' workflows and where misalignments or challenges emerged.
(2) **Systemic Contradictions:** challenges that developers who are visually impaired faced, particularly those arising from systemic *contradictions.* For example, some participants struggled with context switching imposed by the AI, while others had difficulty navigating Copilot's dynamic behaviors, which disrupted the sequential workflows that are critical for developers who are visually impaired. These disruptions required additional navigation adaptations to maintain productivity.
(3) **Adaptive Solutions:** possible adaptive solutions to the challenges we identified. By applying the Activity Theory framework, we gained insights into how AI assistants could be better aligned with the specific workflows of developers who are visually impaired, ultimately fostering more inclusive and efficient coding environments.

By integrating Activity Theory into our thematic analysis, we gained a deeper understanding of how developers who are visually impaired engage with AI coding assistants. This approach allowed us to identify key accessibility themes and design elements of Copilot that either supported or hindered participants' work processes. Our findings contribute to the broader goal of rethinking accessibility paradigms in AI-assisted coding environments.

4 Findings

Through our interviews and observations, we identified key themes that highlight the complex relationship between AI coding assistants, accessibility needs, and coding practices. These themes address our research question by revealing the challenges and opportunities that developers who are visually impaired encounter when using AI-assisted coding tools like GitHub Copilot.

In this section, we present these themes. For each theme, we include illustrative quotes and describe critical incidents from our interviews. These incidents provide insight into the lived experiences of our participants, offering concrete examples of how generative AI coding assistants can both empower and hinder developers who are visually impaired. We frame our findings within the Activity Theory framework to contextualize these challenges as systemic misalignments between users, their tools, and the coding environments they work in. This approach allows us to examine how AI coding assistants mediate the development process and where breakdowns occur, helping us identify opportunities for more accessible and inclusive AI coding assistants.

4.1 RQ1: Challenges and Opportunities of AI Coding Assistants for Developers Who Are Visually Impaired

4.1.1 ***AI and Control***. Our interviews revealed that AI coding assistants contributed to a dynamic sense of control among developers who are visually impaired. This enhanced control manifested in several positive ways. Participants described experiencing a shift toward strategic control over the coding process. Rather than manually performing every task, they found that AI assistant allowed them to delegate routine or less engaging tasks, enabling them to focus on high-level decision-making and overall code structure. This ability to offload work gave them greater control in shaping their development process while maintaining oversight of their projects. For instance, P7 highlighted how Copilot eased their workload and gave them more control over their coding process by handling tasks they found tedious, such as generating docstrings—special multi-line comments in programming that explain the functionality of a function, class, or module:

> *"Overall, my experience with Copilot is positive, especially because it helps me with tasks I don't enjoy. I will write the majority of my own code [...], but where it [Copilot] can help me, I'll definitely let it [Copilot] do it. I love having it generate docstrings for me. That's cool, because frankly, I don't like writing documentation. I'd rather code and let it do the heavy lifting for me..."* - P7, Developer with no vision.

This experience demonstrates how AI empowers developers by taking over tedious tasks they prefer to avoid. By automating routine work, AI enables developers to maintain greater control over their workflow and focus on more complex and engaging coding challenges. Similarly, participant P1, who had prior experience with AI coding assistants, compared coding with AI to piloting an aircraft, emphasizing how these assistants increase their sense of control:

> *"I sort of compare it [Copilot] to pair programming almost, where [...] I'm sort of driving, but the other person is doing all the sort of drudgery work. And what I'm really mostly doing now is almost like what a pilot does when they're flying a plane. They're [...] not flying the plane physically for most of it. Most of it, the plane's flying itself, but the pilot does have to keep an eye on. Are we still going in the right direction? What's that big thing coming towards us really fast? Should we avoid that?"* - P1, Developer with no vision.

From the perspective of Activity Theory, these findings illustrate the transformative role of AI tools as mediators in the coding process. In this context, Copilot functions as an intermediary that actively shapes a developer's control over their work, allowing developers to reallocate their focus toward higher-level strategic decisions, as demonstrated by P7 and P1.

This shift in control also aligns with the concept of *supervisory control* [24, 87], in which humans oversee and direct automated systems rather than executing tasks manually. In this case, the developers who are visually impaired guide the AI coding assistants by monitoring their outputs, making corrections when necessary, and ensuring alignment with their broader coding goals. Just as a pilot monitors and manages an aircraft's automated systems while remaining responsible for high-level navigation and safety decisions, developers can delegate routine implementation details to Copilot while maintaining oversight and control over the overall project direction. This dynamic shifts the developer's role from manually writing every line of code to curating, refining, and strategically guiding the AI's output.

In this new paradigm, control manifests as the ability to guide the overall direction of the code, make critical decisions, and intervene when necessary. While the developer retains ultimate authority over the coding process, the nature of that control shifts to a more abstract and strategic level. Rather than focusing solely on writing individual lines of code, developers can now oversee and direct AI-generated suggestions to ensure they align with the broader project architecture and goals.

However, this shift in control also requires developers to develop new skills, such as crafting effective prompts to generate useful AI outputs, quickly assessing the quality and relevance of the AI-generated code, and maintaining a high-level understanding of the overall software design. As highlighted by P1:

> *"...when I was checking that initial version of the day counting function, I wasn't sure if it [Copilot] realized that it wasn't always going to be the same year, and it [Copilot] didn't realize that in this case. But I've seen it make that kind of mistake before [...] You just need to be careful with your prompts [communication with the AI assistant]."* - P1, Developer with no vision.

This shift in AI-assisted coding, where developers focus on strategic oversight rather than manually completing every coding task, marks an evolution in how developers who are visually impaired maintain control over their coding workflows [8]. The challenge lies in designing AI coding environments that empower developers with this level of control while ensuring accessibility.

To achieve this, designers of AI coding environments must recognize that the needs of developers who are visually impaired often differ from those of sighted developers [8, 27, 107]. While sighted developers frequently rely on visual cues and dynamic, exploratory interfaces, visually impaired developers usually require predictable, structured interactions with clear, interpretable feedback [63]. This requirement is not merely a preference but a necessity that allows them to effectively understand, navigate, and maintain control over their workflow [48, 62]. Now, a key consideration in designing AI coding environments for visually impaired developers is enabling them to anticipate the AI's actions and seamlessly integrate its assistance into their coding tasks [64]. However, generative AI inherently introduces a degree of unpredictability, making this integration complex. As a result, developing AI-driven coding environments that balance strategic oversight with accessibility is not a trivial task and requires careful, thoughtful design.

4.1.2 ***Context Switching Difficulties in AI-assisted Interfaces***. Although Copilot provided a sense of control and empowerment, it also introduced new challenges related to context switching. Our findings highlight that the dynamic nature of AI-generated suggestions and the frequent need to shift focus between writing code and reviewing AI outputs created disruptions. For developers who are visually impaired, this shift interfered with the structured navigation strategies they had developed for traditional coding environments [8], making it harder to maintain workflow continuity.

The AI assistant frequently triggered unexpected view changes, forcing developers to shift their focus to different tasks or sections of the coding environment. For example, as developers interacted with Copilot's suggestions, the system would automatically switch their view to the newly generated code. This disrupted their workflow, especially because they were not always notified about these sudden shifts. As a result, developers struggled to maintain context, making it even more challenging to navigate and integrate AI-generated code seamlessly.

The challenges of context switching in AI-assisted coding environments pose a significant accessibility barrier for developers who are visually impaired. These difficulties highlight the need for AI coding assistants to be designed with greater attention to the needs of screen reader users, who rely on sequential navigation [37, 109].

This challenge became evident with P9, who encountered difficulties after typing a question in the embedded chat window. When Copilot generated a response, he struggled to locate and navigate to the text where the answer was displayed, disrupting his workflow and adding unnecessary cognitive load:

> *"Okay, so do I need, I'm just, I'm trying to understand if we need to press the up and down [the participant repeatedly pressed keys to locate the AI assistant's response]. Seems like it still, the focus is still on my question that I typed [and not on the AI's response]..."* - P9, Developer with no vision.

P1 emphasized that blind users depend on maintaining a stable and consistent context to work efficiently, largely due to how screen readers are designed. However, the frequent and abrupt context switching introduced by AI coding assistants disrupts this continuity, making it difficult for developers who are visually impaired to stay oriented and manage their coding tasks effectively:

> *"A blind person really only has one thing they can see at any given time. It would be that one window, one line of text, one line of whatever it is [...] because screen readers, that's just how they work. You can only see one thing at a time. They work sequentially and not parallel."* - P1, Developer with no vision.

P5 explained that the context-switching issue worsened due to the many keystrokes required to navigate between different contexts, such as reaching the Copilot chat window after writing code. The need for multiple keystrokes or clicks for minor transitions caused him to lose track of his original task, leading to frustration. This additional effort further amplified the challenge, making it harder to maintain workflow and stay focused on coding tasks:

> *"...when I'm coding, I have to memorize so much... It's a lot of, I'm going to reference this file again because I can't 100% remember what I said...I was going to do. And you know when I'm going to Copilot, when I'm pressing F6 a couple of times and I'm pressing tab a couple of times, that creates a little frustration almost, because I want to get back to the task [The participant used multiple key presses to navigate to the Copilot chat window and then return to their code]. And that frustration makes it hard to remember what I was thinking about, to begin with. And then...I forget the context...now I have to go back and check again."* - P5, Developer with no vision.

The challenges of context switching in AI-assisted coding environments were further exacerbated by unexpected AI responses that caused sudden and confusing shifts in context. P5, a developer with no vision, described a particularly frustrating experience where merely navigating through the coding interface unintentionally triggered AI actions, making navigation even more difficult:

> *"Did I accidentally just accept [accept the AI's suggestion] I don't, I think I hit tab. See, this is part of the problem too. It's so easy to accidentally accept a suggestion because when you move the cursor, Copilot immediately is like, oh, I have a suggestion for this and I'm just going to suggest it to you. But you know, again, the way a lot of us [developers who are visually impaired] navigate the UI is usually with the tab key. And so it's so easy to just accidentally insert a suggestion"* - P5, Developer with no vision.

Similarly, P2 highlighted a significant challenge for users who rely on screen magnification: the AI assistant sometimes displayed information in areas of the screen that were out of view. Because these users were zoomed into a specific section of the screen, they often remained unaware when the AI assistant provided suggestions, as the content appeared outside their visible area:

> *"...If I, for example, zoom in like this [the participant zoomed into a specific screen area], I often don't get any information if there's some programming error or something going on in this area [the participant pointed to another part of the screen]. When I look at this [the zoomed-in area] and I just remember every time I have to go this way [The participant pointed to the unseen screen area where Copilot placed suggestions.] Sometimes I don't see that [AI suggestion] if I zoom in. So for people who are using Zoom, it will be better that it [AI assistant] will show up in the middle of the program because I don't see it [AI assistant] when it's in this area [zoomed-in area of the screen]. If there's something I have to install, for example, to run anything."* - P2, Developer with low vision.

These participant experiences align with and extend prior research on context-switching challenges for developers who are visually impaired. Our findings build upon the work of Albusays et al. [6], who conducted interviews and observations with blind software developers to examine code navigation difficulties in traditional IDEs. Their study identified the challenges blind developers face when moving between different sections of a program. Our research expands on these insights by showing how AI-assisted coding environments amplify these difficulties by introducing additional interface elements and new interaction modes, further complicating navigation and workflow continuity. While Albusays et al. [6] focused on traditional IDEs, our study reveals that AI-assisted coding environments introduce an additional layer of complexity by requiring developers to frequently switch contexts within the same application. This added complexity exacerbates the already challenging task of code navigation for developers who are visually impaired. The dynamic nature of AI-generated suggestions and the frequent need to shift focus between manually written code and AI-generated content disrupt the mental models and navigation strategies that blind developers have developed.

From an Activity Theory perspective, the challenges of context switching reveal *contradictions* in the interaction between the user and the AI coding assistant. These *contradictions* likely arise because AI coding assistants are primarily designed for visual interaction, which conflicts with the sequential, nonvisual navigation methods used by screen reader users. However, several additional factors contribute to this misalignment:

- The need for multiple keystrokes to move between different contexts within the AI-assisted environment.
- Unexpected AI behaviors that interfere with established navigation patterns of developers who are visually impaired.
- Information being displayed in inaccessible areas due to screen magnification.

This mismatch between the tool's design and the needs of developers who are visually impaired disrupts their ability to stay focused and maintain a smooth workflow.

4.1.3 ***AI Timeouts.*** While AI coding assistants gave developers greater control over their code and reduced the need for manually completing every task, they also introduced cognitive challenges. Participants specifically expressed a need for moments of disconnection from the AI. The constant stream of AI-generated suggestions—especially the frequent appearance of *ghost text*—created a trade-off between maintaining control and mental focus. This tension led to a key theme in our study: participants' desire for what we call *"AI timeouts"*—periods of uninterrupted coding without AI intervention. For instance, participant P1, a braille display user, stressed the importance of having control over when AI suggestions appear. They found that frequent AI interventions disrupted their thought process, stating:

> *"What I usually do when this happens [when ghost text is presented] is I'll turn speech off for a bit so I can think...."* - P1, Developer with no vision.

Participant P1's strategy of turning off speech to think highlights the issue of *information overload*, a challenge explored by Ahmed et al. [3] in their research on non-visual interfaces. Ahmed et al. explain that screen-reader users often struggle to determine the relevance or importance of content without first listening to at least some of it. This necessity frequently results in cognitive overload for users who are visually impaired.

In AI-assisted coding environments, this issue becomes even more pronounced. P1's experience illustrates how the continuous stream of AI-generated suggestions, conveyed through speech output, can overwhelm cognitive capacity. By choosing to "turn speech off for a bit," P1 effectively creates a temporary barrier—an **AI timeout**—allowing them to pause, process information, and make decisions without the constant influx of new suggestions. This aligns with findings by Bigham et al. [16], who suggest that slower-paced interactions can enhance usability for visually impaired users.

The concept of AI timeouts extends beyond merely managing ghost text suggestions. Some participants expressed a need for extended periods of uninterrupted coding without any AI input. For instance, P7 emphasized the importance of having the option to engage in uninterrupted coding sessions, stating:

> *"Well, hey, can we get a mode that says, hey, I just want to get all my code done. I just want to write code. I don't care right now. I don't care how buggy it is right now. I really don't care. But I want to stay in the IDE because when I'm done, I'm going to use a shortcut and I'm going to go through, I'm going to find all my errors and I'll fix them, but I don't want it [AI assistant] to get in the way [...] But let me just get my thoughts out. "* - P7, Developer with no vision.

This experience highlights the potential for cognitive dissonance when AI assistance operates at a faster pace than the developer's thought process. Although the AI's suggestion may have been useful, it momentarily disrupted P7's train of thought, creating a mismatch between the AI's proactive assistance and the participant's cognitive workflow. While P7's experience underscores the need for AI timeouts, it is important to note that not all participants shared this sentiment. P1's reflection, for example, illustrates a different perspective—one where AI acts as a catalyst for learning and expanding problem-solving approaches:

> *"I think it [AI assistant] was a couple steps ahead of me now and again because I was sort of just expecting, okay, I need to get that date [date requested in their coding task], and then I'm going to need to somehow get a year later out of it. And I didn't actually really consider that you'd have to sort of parse that date and convert it into an actual date first. [...] So when it [AI assistant] made a constructor and just transformed it into month, day, year, I'm like, okay, well, that's not immediately what I had in mind, but on second thought, that does make a lot of sense. [...] I probably would have done that as a next step myself, but it just sort of preempted me on there."* - P1, Developer with no vision.

This experience aligns with Vygotsky's concept of the Zone of Proximal Development [102], which suggests that learning is most effective when guidance bridges the gap between what a learner can do independently and what they can achieve with support. In this case, the AI guided P1 toward a more advanced solution. However, maintaining a balance between beneficial cognitive challenges and overwhelming disruptions is crucial. While this instance demonstrated a positive outcome, it highlights the need for AI systems that can dynamically adjust their level of intervention based on the user's expertise and cognitive load. By incorporating adaptive assistance, AI coding assistants can create a more effective and enriching coding experience—one where developers are both supported and challenged in a way that enhances their problem-solving abilities without causing excessive cognitive overload.

Conversely, P4's response to how AI suggestions influenced their planning highlights two key insights. First, AI assistants can encourage developers to adopt best coding practices earlier in the development process. Second, AI timeouts could play a role in giving developers the necessary time to reflect on different aspects of their program before implementing AI-generated suggestions:

> *"...[I would] not have remembered to put those guard clauses in until I got, like, my unit testing hat on [the participant was pointing to AI-generated guard clauses, a best practice for handling edge cases]. I've seen some videos of like, I think Uncle Bob, Bob Martin did some unit testing videos where he literally had a hat that he would flip around and it was two hats put together and he'd be like, coder, unit test, or coder and those kind of things of like, oh, I wasn't there yet [was not thinking about guard clauses yet], but if you [AI assistant] want to help me get there early, that's fine. That's great."* - P4, Developer with low vision.

This reflection highlights how AI can blur the traditional boundaries between different coding phases, potentially reducing the need for certain AI timeouts. In this case, the AI's proactive suggestions for guard clauses eliminated the need for a "timeout" between the coding and testing phases by integrating best practices early in the development process. However, this scenario also emphasizes the importance of "flexible AI timeouts". While P4 valued the early inclusion of guard clauses, other developers might prefer to maintain distinct mental phases in their coding workflow. In this context, AI timeouts do not necessarily mean disengaging from AI assistance entirely but rather enabling developers to control when and how AI interventions occur. By allowing developers to schedule AI timeouts or receive advanced suggestions at specific points in their workflow, AI-assisted coding tools can accommodate different coding styles and personal preferences. This flexibility could be particularly beneficial for developers who are visually impaired, as they may have structured routines or mental models for managing distinct phases of the coding process [42, 50]. Providing adaptive AI interaction could enhance productivity without disrupting established workflows.

P3's experience further illustrates the need for a balanced approach to AI timeouts, ensuring that AI assistance is helpful without becoming disruptive:

> *"Honestly, I think it's [AI assistant] super helpful because like, I'm going to be honest, like you were saying before, I always forget datetime manipulation. [...] So having the ability to just say what I want, have it [AI assistant] understand the context of the programming language I'm in and help me kind of get there quicker without having to kind of start searching around the web, figuring out the reading documentation again for the datetime methods. Like I have to all the time."* - P3, developer with low vision.

As the quote illustrates, developers may choose to keep the AI when it provides helpful support, such as recalling correct syntax or assisting with complex code manipulations. However, for tasks requiring deep problem-solving or creative thinking, they may prefer to use AI timeouts to independently work through solutions.

The concept of "AI timeouts" reflects *contradictions* within the activity system, arising from misalignments between the *subject* (developers), the *object* (coding tasks), and the *tool* (Copilot). While Copilot serves as a mediating artifact designed to facilitate coding, its dynamic and sometimes intrusive interventions create tensions that can disrupt developers' workflows. This contradiction likely arises because the tool's design assumes that continuous AI assistance improves productivity, whereas developers might actually need uninterrupted periods to process and structure information. These tensions highlight the necessity of adaptable AI systems that calibrate their level of intervention based on the user's needs.

4.1.4 *AI-Assisted Coding Efficiency*.

While our study highlighted challenges like context switching, it also revealed key benefits of AI coding assistants, particularly in enhancing coding efficiency for developers who are visually impaired. This efficiency improvement stemmed from two main factors:

(1) **Copilot's Proactive Code Generation**, which anticipated and generated relevant code, reducing manual effort.
(2) **Copilot as an Accessible, Always-Available Coding Partner**, providing instant non-judgmental support.

Proactive Code Generation. Participants highlighted how the AI assistant boosted their productivity by anticipating their needs—proactively generating code that resolved existing issues or introduced features they had not initially considered. For instance, P7 described a moment when the AI assistant demonstrated foresight by automatically parsing and transforming a date into a usable format. Upon reflection, P7 realized that this step aligned perfectly with their next intended action, streamlining their workflow and reducing the need for manual adjustments:

> *"It [AI assistant] was a couple steps ahead of me. [The participant's coding task involved calculating the number of days until a birthday provided by an end-user.] I was initially focused on getting the date and calculating a year later, but I didn't actually really consider that you'd have to sort of parse that date [date given by the end-user] and convert it into an actual date first [converting to an actual date was something the generative AI did for them]. That makes perfect sense in hindsight [...] So when it [AI assistant] made it a constructor and just transformed it into month, day, year, I'm like, okay, well, that's not immediately what I had in mind, but on second thought, that does make a lot of sense. [...] I probably would have done that as a next step myself, but it just sort of preempted me on there."* - P7, Developer with no vision.

Similarly, P1 emphasized their appreciation for the AI's ability to proactively generate code, particularly in how it anticipated their next steps and aligned with their intentions as a developer:

> *"I'm gonna try something else. I'm actually very curious [...] I'll auto modify that [The participant used a Copilot command to 'auto-modify' their code, allowing the AI to proactively edit and expand their initial work]. That sounds good [The participant was reviewing the code generated by the AI"]. There are days until your next birthday. Yeah, format days until next birthday [The participant continued reviewing the code generated by the AI.] That [code generated by the AI] is scarily good. Actually that [AI generated code] is exactly what I was going to do."* - P1, Developer with no vision.

Prior research has documented the challenges that developers with visual impairments face when navigating and understanding code structures in traditional development environments [5, 11, 78]. Given these challenges, we argue that AI's proactive code generation holds potential for this population. Developers who are visually impaired often need to traverse multiple documentation pages or code files to construct appropriate solutions—a process that is not only time-consuming but also more prone to errors when relying on screen readers [6, 20]. In this context, AI's ability to predict coding needs and generate relevant code snippets is especially valuable, as it can reduce cognitive load and enhance efficiency for developers who are visually impaired.

AI as an Accessible, Always-Available Coding Partner. Some participants described the AI assistant as a supportive 'buddy' that guided them through their coding tasks, providing assistance and enhancing their productivity. They appreciated having an AI-powered companion they could rely on for coding assistance, providing quick answers and relevant code snippets when needed. For example, Participant P3 likened the AI assistant to a knowledgeable colleague who is readily available to offer solutions and assist with coding challenges:

> *"[with the AI coding assistant] you have that kind of buddy to kind of ask real quick, you know, what was that? You know, what was the daytime method that I needed to use to convert that particular string into the right thing? [The participant refers to asking the AI about functions and methods needed for the date-related coding task]"* - P3, Developer with low vision.

All participants agreed that the AI assistant was beneficial in helping them complete their software development tasks. However, the degree of reliance on the assistant varied among individuals. Some, like P6, depended on it extensively, using it to complete nearly their entire assigned workload:

> *"I just copy and pasted the whole task [instructions for a coding task assigned in our study] into chat [Copilot's chat interface], and it [AI assistant] gave me the whole implementation that we had to do. Very, very minor tweaks left to do. I mean, the date formats worked out, right? [The participant was assigned a coding task involving date formats.] Essentially we just added comments and tinkered with the code [...] But otherwise, it [AI assistant] just did everything for us."* - P6, Developer with low vision.

Even participants with limited experience using AI assistants, such as P9, acknowledged its potential to accelerate coding tasks. Their experience in our study sparked greater interest in exploring the technology further:

> *"I would love to try and explore more [about AI coding assistants]. But, it [the AI coding assistant] has definitely a potential [...] just doing quick work, especially writing quick functions and doing also all those tasks [...] It's [the AI assistant is] bringing a lot of time saving."* - P9, Developer with no vision.

For developers who are visually impaired, this aspect of AI assistance could be particularly valuable, as it offers constant, non-judgmental support without requiring face-to-face interaction. This can be especially beneficial in workplace environments where they may feel hesitant to frequently ask colleagues for help [6]. Generative AI assistants have the potential to make software development more accessible and efficient, enabling developers to work more independently without relying on coworkers for assistance.

From the perspective of Activity Theory, Copilot serves as a mediating artifact that reshapes the coding activity by shifting its role from a passive tool to an active collaborator. Within the framework of Activity Theory, the interaction between the *subject* (developer), *object* (coding task), and *tool* (AI assistant) is no longer a linear process but a dynamic, adaptive exchange. Instead of developers following a rigid, step-by-step workflow, Copilot enables a more fluid interaction where AI-generated suggestions proactively shape the problem-solving process. Acting as a readily available and non-judgmental coding companion, Copilot can function as a "buddy" that developers can turn to at any moment to clarify uncertainties, propose solutions, and reduce cognitive load. By integrating AI as an active collaborator within the activity system, Copilot can help developers navigate complex programming tasks with greater confidence and independence.

5 Discussion

Our findings show that AI coding assistants provide powerful capabilities, new opportunities, and greater control for developers who are visually impaired, but they also introduce new accessibility challenges. Issues such as context switching and managing AI-generated suggestions highlight the need for a new approach to accessibility in AI coding environments. Building on Nielsen's argument that generative AI has introduced a new paradigm for HCI [68] and the design principles for generative AI tools outlined by Weisz et al. [104], we propose design recommendations to ensure AI coding assistants are accessible and beneficial for all developers.

5.1 AI Control and Context-Switching Management in AI Coding Assistants

Our study highlights the mixed experiences of developers who are visually impaired when using AI coding assistants. While these assistants empower developers by allowing them to focus on higher-level strategic thinking, they also introduce new challenges. Participants, such as P5 and P9, reported difficulties in navigating between their static code and the dynamic windows used for AI interactions. This constant context switching led to frustration, disrupting their workflow and breaking their cognitive focus.

Previous research highlights that the visually oriented nature of Integrated Development Environments (IDEs) presents challenges for developers who are visually impaired, often leading to a loss of control [6, 22, 96]. P1 also noted that screen readers require users to read code sequentially, making context switching particularly difficult [5–7, 59, 90]. Several tools, such as StructJumper and CodeTalk, have been designed to aid navigation in traditional static coding environments [10, 11, 77, 78]. However, these tools primarily focus on static code elements. In contrast, AI-assisted coding environments introduce new challenges by incorporating dynamic code elements [73]. Our findings reveal that developers who are visually impaired must now manage the additional complexity of interacting with AI-generated suggestions, making context switching even more difficult. Navigating between static code and AI-generated content requires frequent shifts in focus across different interface elements [66, 73]. Developers must assess their own code, review AI-generated suggestions, and interact with various UI windows required for AI engagement [105]. This continuous back-and-forth process can increase cognitive load, forcing developers to constantly evaluate AI outputs while maintaining an understanding of their overall code structure [84]. As a result, the interplay between AI assistance and manual coding can fragment attention, disrupt workflow continuity, and impact productivity, code quality, and trust in the AI-generated content [55, 73, 103].

Our findings highlight the urgent need for accessible tools specifically designed for dynamic coding environments. These tools should aim to streamline workflows, minimize AI disruptions, and provide better support for developers who are visually impaired. By addressing the challenges of context switching and maintaining control, such tools can help developers navigate AI-assisted coding environments more effectively and improve overall accessibility.

5.1.1 Addressing the Research Gap. Our research revealed both new accessibility opportunities and challenges in AI coding assistants for developers who are visually impaired. On one hand, AI coding assistants empowered developers by shifting their role toward supervisory control over the code. However, these benefits were accompanied by significant challenges, especially related to context switching, which disrupted their workflow and made navigation more complex.

While previous studies [6, 22, 96] have identified accessibility barriers in traditional IDEs—such as visually-oriented interfaces and linear screen reader navigation for static code [5, 7, 59, 90]—our findings reveal that AI-driven coding environments exacerbate these issues. The dynamic nature of AI-assisted coding introduces additional complexities that amplify accessibility barriers. Existing tools like StructJumper and CodeTalk [10, 11, 77, 78], designed for static code, may not fully support AI-driven workflows. Furthermore, prior research on AI-assisted coding has examined cognitive strain and productivity for general developers [55, 66, 73, 105]. But, our work introduces an accessibility perspective. Frequent context switching and fragmented workflows present significant challenges for developers who are visually impaired, emphasizing the urgent need for AI tools designed with accessibility at their core to ensure equitable participation in evolving coding practices.

5.1.2 Design Recommendations. Based on our findings, we propose a set of design recommendations aimed at improving interaction continuity, personalization, and information management in AI-assisted coding environments. These recommendations build on established principles such as *Design for Generative Variability* by Weisz et al. [104], which emphasizes the importance of visualizing the user's journey to accommodate diverse needs. From an Activity Theory perspective, our recommendations seek to realign the AI coding assistant (*tool*) with developers who are visually impaired (*subject*) to help them successfully complete coding tasks (*object*).

First, we recommend implementing a **context history log** [21] to mitigate workflow disruptions caused by AI-driven context switching. This feature would provide a sequential record of user interactions with the AI, allowing developers to review and revisit previous steps in their coding process. For visually impaired users, a structured history log could be especially valuable for maintaining workflow continuity and recovering from unexpected focus shifts.

Additionally, this log could enhance the system's adaptability by learning from user behavior. For example, if a developer frequently accesses the inline chat for code explanations, the AI could prioritize and streamline access to this feature, reducing cognitive load and improving efficiency. By addressing workflow *contradictions*, a context history log would ensure that developers maintain control over their interactions with the AI, making AI-assisted coding environments more predictable and user-friendly.

Second, we propose integrating an **interaction organizer** to help mitigate the challenges caused by the sequential navigation constraints of screen readers. Unlike sighted developers, who can visually scan and filter information quickly, developers who are visually impaired must process content in a linear fashion, which can make it difficult to efficiently navigate AI-generated suggestions [20]. The interaction organizer would allow users to structure AI-generated suggestions more effectively by enabling them to:

- Group related AI suggestions into folders or categories.
- Assign meaningful labels and keywords for quick retrieval.
- Attach personal notes or comments to AI-generated code.

By incorporating these capabilities, the interaction organizer would help developers streamline their workflow, making it easier to manage and retrieve AI-generated outputs while reducing cognitive strain. From an Activity Theory perspective, this feature helps resolve *contradictions* between the dynamic and non-linear nature of AI-generated suggestions and the sequential workflows that visually impaired developers rely on. By providing structured ways to categorize and personalize AI outputs, the interaction organizer improves the tool's role as a mediator in the coding process, making AI-assisted development more efficient and accessible.

5.2 The Need for "AI Timeouts" in AI-Assisted Coding

Our study uncovered a complex relationship between increased productivity and cognitive overload in AI-assisted coding environments for developers who are visually impaired. While participants reported productivity gains—such as AI proactively anticipating their coding needs—these benefits also introduced a challenge: the need for *AI timeouts*. These timeouts represent intentional breaks from AI interventions, allowing developers to manage information overload and maintain focus during their workflow.

The continuous stream of AI-generated suggestions, particularly inline *ghost text*, seemed to have contributed to cognitive overload among participants. This overload created a need for *AI timeouts*—temporary pauses in AI interventions to allow for uninterrupted focus and reduce mental strain.

Activity Theory provides a useful framework for understanding this challenge, as it reveals a *contradiction* between the AI assistant's intended role of enhancing productivity and its unintended effect of disrupting focus. This finding aligns with prior research on information overload in visual interfaces [3, 32], which shows that excessive information can overwhelm users, particularly those who are visually impaired. Our study extends this research by demonstrating how AI-generated content can intensify cognitive overload in AI-assisted coding environments. This underscores the importance of designing AI interactions that are more controlled and customizable, ensuring that developers who are visually impaired can maintain focus and effectively manage their workflows.

The need for *AI timeouts* varied among participants, highlighting the complexity of designing AI coding assistants for developers who are visually impaired. While some participants valued proactive AI assistance, others wanted greater control over when and how AI intervened in their coding process. This variation underscores the importance of flexible, personalized customization rather than a one-size-fits-all approach, which is often emphasized in existing research on accessible development tools [10, 11, 77, 78].

Building on Gajos et al.'s *adaptive interfaces* concept [31], our findings suggest that AI-assisted coding environments could benefit from adaptive features. From an Activity Theory perspective, these adaptive interfaces help resolve *contradictions* by aligning AI mediation with user goals and preferences. Specifically, AI coding assistants could learn individual preferences for AI intervention, dynamically adjusting the frequency and type of AI suggestions based on task complexity. Additionally, providing easily accessible controls for enabling and disabling AI assistance would grant developers greater autonomy over their workflow. This approach ensures that AI remains a supportive tool rather than a source of cognitive overload.

5.2.1 *Addressing the Research Gap.*

Our research introduces *AI timeouts* as a strategy to mitigate cognitive overload for developers who are visually impaired using AI coding assistants. Prior work highlights AI's ability to enhance productivity by automating repetitive tasks and offering anticipatory suggestions [4, 70, 106]. However, our findings reveal new challenges, such as cognitive strain from continuous AI-generated inputs like inline *ghost text*. Building on studies of information overload for visually impaired users [3, 32, 73, 84], we extend this discussion to dynamic AI-driven environments, where proactive AI mediation can disrupt cognitive coherence [73, 84]. Using Activity Theory, we frame this tension as a *contradiction* between AI assistance and focus disruption, proposing AI timeouts as a novel accessibility strategy to balance intervention and cognitive load.

5.2.2 *Design Recommendations.*

To improve the accessibility and usability of AI coding assistants for developers who are visually impaired, we recommend design strategies aligned with the *Design for Mental Models* principle by Weisz et al. [104]. This principle emphasizes the importance of aligning AI behavior with the user's expectations to balance supportive AI intervention with user autonomy. One of our recommendation is to allow developers to **customize the AI's behavior** to match their mental model. For instance, tools could offer adjustable modes such as "Low Detail" and "High Detail," enabling users to control the level of information the AI provides based on task complexity or cognitive load. This flexibility ensures that AI assistance aligns with individual problem-solving preferences, reducing unnecessary cognitive strain. Customization can also resolve *contradictions* between the user's mental model and the often unpredictable nature of AI-generated suggestions, which may sometimes provide excessive or distracting information.

Another essential feature that we propose is the implementation of **AI timeouts**, which serve two key purposes:

- **Preserving Autonomy**: Timeouts allow developers to temporarily pause AI interventions, giving them space to reflect and solve problems independently. This can prevent over-reliance on AI and help maintain control over the coding process. From an Activity Theory perspective, these timeouts address *contradictions* by giving users greater control over the tool, ensuring AI assistance aligns with their goals and cognitive needs.
- **Enhancing Learning**: By pausing AI interventions, developers have the opportunity to engage with their own problem-solving strategies. This reinforces their mental model, strengthens coding skills, and promotes deeper understanding. This aligns with Activity Theory's emphasis on fostering user agency, allowing developers to actively shape their interactions with Copilot.

For developers who are visually impaired, maintaining a clear and consistent mental model of the coding environment can be essential for effective work. Customizable AI behavior and strategic pauses could help ensure interactions remain predictable and aligned with user expectations, fostering a more intuitive and personalized experience. By supporting mental model alignment, these design features bridge the gap between user cognition and AI functionality, ultimately improving accessibility and productivity.

5.3 Limitations and Future Work.

Our study provides valuable insights into the experiences of developers who are visually impaired using AI coding assistants. But it also has limitations. We interviewed a small group of 10 male participants, which restricts the diversity and generalizability of our findings. Studying emerging technologies like GitHub Copilot and recruiting a niche population, such as developers who are visually impaired, often results in a limited and homogeneous sample. However, despite participants' varying levels of experience with AI coding assistants, their shared challenges and opportunities provide foundational insights into how these tools impact accessibility.

Additionally, our study focused exclusively on GitHub Copilot, meaning we did not explore other AI coding assistants that may have different features and accessibility considerations. Future research should examine a broader range of AI coding tools to gain a more comprehensive understanding of accessibility challenges across different platforms. Another limitation is that participants had limited prior exposure to GitHub Copilot, meaning our findings may not fully capture the long-term benefits and challenges of using AI coding assistants. Longitudinal studies are needed to investigate how developers who are visually impaired adapt to AI-assisted environments over time, as well as any lasting accessibility barriers or professional impacts that may emerge.

6 Conclusion

We examined the experiences of 10 developers who are visually impaired as they interacted with an AI coding assistant. Our study highlights the dual impact of AI coding assistants on developers who are visually impaired. While these tools enhance control and provide valuable support, they also introduce new accessibility challenges and cognitive demands. Participants' experiences underscore the need to rethink accessibility in AI-driven coding environments. As AI continues to reshape software development, the design decisions made today will determine the inclusivity of the tech industry for years to come. Our findings call on researchers, designers, and industry leaders to make accessibility a priority in the development of next-generation AI tools, ensuring they empower all developers and foster equal opportunities in the field.

ACKNOWLEDGMENTS. Special thanks to all the anonymous reviewers who helped us to strengthen the paper as well as the software developers who participated in the study. This work was partially supported by NSF grants 2339443 and 2403252.

References

[1] Shadi Abou-Zahra, Judy Brewer, and Michael Cooper. 2018. Artificial intelligence (AI) for web accessibility: Is conformance evaluation a way forward?. In *Proceedings of the 15th international web for all conference*. 1–4.

[2] Rudaiba Adnin and Maitraye Das. 2024. " I look at it as the king of knowledge": How Blind People Use and Understand Generative AI Tools. In *Proceedings of the 26th International ACM SIGACCESS Conference on Computers and Accessibility*. 1–14.

[3] Faisal Ahmed, Yevgen Borodin, Yury Puzis, and IV Ramakrishnan. 2012. Why read if you can skim: towards enabling faster screen reading. In *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*. 1–10.

[4] Humaid Al Naqbi, Zied Bahroun, and Vian Ahmed. 2024. Enhancing work productivity through generative artificial intelligence: A comprehensive literature review. *Sustainability* 16, 3 (2024), 1166.

[5] Khaled Albusays and Stephanie Ludi. 2016. Eliciting programming challenges faced by developers with visual impairments: exploratory study. In *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*. 82–85.

[6] Khaled Albusays, Stephanie Ludi, and Matt Huenerfauth. 2017. Interviews and observation of blind software developers at work to understand code navigation challenges. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*. 91–100.

[7] Khaled L Albusays. 2020. *The Role of Sonification as a Code Navigation Aid: Improving Programming Structure Readability and Understandability For Non-Visual Users*. Rochester Institute of Technology.

[8] Nasser Ali Aljarallah and Ashit Kumar Dutta. 2024. A Systematic Review on Developing Computer Programming Skills for Visually Impaired Students. *Journal of Disability Research* 3, 2 (2024), 20240018.

[9] Humberto Lidio Antonelli, Leonardo Sensiate, Willian Massami Watanabe, and Renata Pontin de Mattos Fortes. 2019. Challenges of automatically evaluating rich internet applications accessibility. In *Proceedings of the 37th ACM International Conference on the Design of Communication*. 1–6.

[10] Ameer Armaly, Paige Rodeghero, and Collin McMillan. 2018. Audiohighlight: Code skimming for blind programmers. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 206–216.

[11] Catherine M Baker, Lauren R Milne, and Richard E Ladner. 2015. Structjumper: A tool to help blind programmers navigate and understand the structure of code. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 3043–3052.

[12] Mark S Baldwin, Jennifer Mankoff, Bonnie Nardi, and Gillian Hayes. 2020. An activity centered approach to nonvisual computer interaction. *ACM Transactions on Computer-Human Interaction (TOCHI)* 27, 2 (2020), 1–27.

[13] Saidarshan Bhagat, Padmaja Joshi, Avinash Agarwal, and Shubhanshu Gupta. 2024. Accessibility evaluation of major assistive mobile applications available for the visually impaired. *arXiv preprint arXiv:2407.17496* (2024).

[14] Jeffrey P Bigham, Chandrika Jayant, Hanjie Ji, Greg Little, Andrew Miller, Robert C Miller, Robin Miller, Aubrey Tatarowicz, Brandyn White, Samual White, et al. 2010. Vizwiz: nearly real-time answers to visual questions. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*. 333–342.

[15] Jeffrey P Bigham, Raja Kushalnagar, Ting-Hao Kenneth Huang, Juan Pablo Flores, and Saiph Savage. 2017. On how deaf people might use speech to control devices. In *Proceedings of the 19th international ACM SIGACCESS conference on computers and accessibility*. 383–384.

[16] Jeffrey P Bigham, Richard E Ladner, and Yevgen Borodin. 2011. The design of human-powered access technology. In *The proceedings of the 13th international ACM SIGACCESS conference on Computers and accessibility*. 3–10.

[17] Jeffrey P Bigham, Irene Lin, and Saiph Savage. 2017. The Effects of" Not Knowing What You Don't Know" on Web Accessibility for Blind Web Users. In *Proceedings of the 19th international ACM SIGACCESS conference on computers and accessibility*. 101–109.

[18] Susanne Bodker. 1989. A human activity approach to user interfaces. *Human-Computer Interaction* 4, 3 (1989), 171–195.

[19] Susanne Bodker. 2021. *Through the interface: A human activity approach to user interface design*. CRC Press.

[20] Yevgen Borodin, Jeffrey P Bigham, Glenn Dausch, and IV Ramakrishnan. 2010. More than meets the eye: a survey of screen-reader browsing strategies. In *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*. 1–10.

[21] Hee Eon Byun and Keith Cheverst. 2004. Utilizing context history to provide dynamic adaptations. *Applied Artificial Intelligence* 18, 6 (2004), 533–548.

[22] Yoonha Cha, Victoria Jackson, Isabela Figueira, Stacy Marie Branham, and André Van der Hoek. 2024. Understanding the Career Mobility of Blind and Low Vision Software Professionals. In *Proceedings of the 2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering*. 170–181.

[23] Khansa Chemnad and Achraf Othman. 2024. Digital accessibility in the era of artificial intelligence—Bibliometric analysis and systematic review. *Frontiers in Artificial Intelligence* 7 (2024), 1349668.

[24] Mark Chignell, Lu Wang, Atefeh Zare, and Jamy Li. 2023. The evolution of HCI and human factors: Integrating human and artificial intelligence. *ACM Transactions on Computer-Human Interaction* 30, 2 (2023), 1–30.

[25] Chris Creed and Sayan Sarcar. 2024. Voice coding experiences for developers with physical impairments. *Journal of Enabling Technologies* 18, 4 (2024), 265–275.

[26] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Zhen Ming Jack Jiang. 2023. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software* 203 (2023), 111734.

[27] Md Ehtesham-Ul-Haque, Syed Mostofa Monsur, and Syed Masum Billah. 2022. Grid-coding: An accessible, efficient, and structured coding paradigm for blind and low-vision programmers. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–21.

[28] Yrjö Engeström. 1987. An activity-theoretical approach to developmental research. *Helsinki: Orienta-Konsultit* (1987).

[29] Y Engeström. 1999. Activity theory and individual and social transformation. *Perspectives on activity theory/Cambridge University Press* (1999).

[30] Claudia Flores-Saviaga, Shangbin Feng, and Saiph Savage. 2022. Datavoidant: An ai system for addressing political data voids on social media. *Proceedings of the ACM on human-computer interaction* 6, CSCW2 (2022), 1–29.

[31] Krzysztof Z Gajos, Jacob O Wobbrock, and Daniel S Weld. 2008. Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. 1257–1266.

[32] Stéphanie Giraud, Pierre Thérouanne, and Dirk D Steiner. 2018. Web accessibility: Filtering redundant and irrelevant information improves website usability for blind users. *International Journal of Human-Computer Studies* 111 (2018), 23–35.

[33] GitHub. 2022. *GitHub Copilot now available for everyone*. https://github.blog/2022-06-21-github-copilot-now-available/ Accessed: 2023-12-06.

[34] GitHub. 2023. GitHub Copilot · Your AI pair programmer. https://github.com/features/copilot Accessed: December 2023.

[35] Cole Gleason, Dragan Ahmetovic, Saiph Savage, Carlos Toxtli, Carl Posthuma, Chieko Asakawa, Kris M Kitani, and Jeffrey P Bigham. 2018. Crowdsourcing the installation and maintenance of indoor localization infrastructure to support blind navigation. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 1 (2018), 1–25.

[36] Cole Gleason, Dragan Ahmetovic, Carlos Toxtli, Saiph Savage, Jeffrey P Bigham, and Chieko Asakawa. 2017. LuzDeploy: A collective action system for installing navigation infrastructure for blind people. In *Proceedings of the 14th International Web for All Conference*. 1–2.

[37] Monica Hegde. 2023. *User Experience Study of Screen Readers for Visually Challenged Users*. Master's thesis. Purdue University.

[38] Shawn Lawton Henry, Shadi Abou-Zahra, and Judy Brewer. 2014. The role of accessibility in a universal web. In *Proceedings of the 11th Web for all Conference*. 1–4.

[39] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[40] Yosef Jabareen. 2009. Building a conceptual framework: philosophy, definitions, and procedure. *International journal of qualitative methods* 8, 4 (2009), 49–62.

[41] Sajed Jalil. 2023. The Transformative Influence of Large Language Models on Software Development. *arXiv preprint arXiv:2311.16429* (2023).

[42] Philip N Johnson-Laird. 1989. Mental models. (1989).

[43] Shaun K Kane, Jeffrey P Bigham, and Jacob O Wobbrock. 2008. Slide rule: making mobile touch screens accessible to blind people using multi-touch interaction techniques. In *Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility*. 73–80.

[44] Victor Kaptelinin and Bonnie A Nardi. 2009. *Acting with technology: Activity theory and interaction design*. MIT press.

[45] Victor Kaptelinin and Bonnie A Nardi. 2012. *Activity theory in HCI: Fundamentals and reflections*. Vol. 13. Morgan & Claypool Publishers.

[46] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–23.

[47] B Kelly, L Phipps, and C Howell. 2005. Implementing a holistic approach to e-learning accessibility [Electronic version]. In *Exploring the frontiers of e-learning: Borders, outposts and migration (ALT-C 2005 12th International Conference Research Proceedings). Retrieved November*, Vol. 27. 2006.

[48] Hourieh Khalajzadeh and John Grundy. 2024. Accessibility of low-code approaches: A systematic literature review. *Information and Software Technology* (2024), 107570.

[49] Mukta Kulkarni. 2019. Digital accessibility: Challenges and opportunities. *IIMB Management Review* 31, 1 (2019), 91–98.

[50] Thomas D LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*. 492–501.

[51] Jonathan Lazar, Alfreda Dudley-Sponaugle, and Kisha-Dawn Greenidge. 2004. Improving web accessibility: a study of webmaster perceptions. *Computers in human behavior* 20, 2 (2004), 269–288.

[52] AN Leóntiev. 1981. Problems of the development of the mind (Published 1931).

[53] Aleksei Nikolaevich Leont'ev. 1978. Activity, consciousness, and personality.

[54] Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. 2025. Prompting Large Language Models to Tackle the Full Software Development Lifecycle: A Case Study. In *Proceedings of the 31st International Conference on Computational Linguistics*. 7511–7531.

[55] 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.

[56] Stephanie Ludi. 2015. Position paper: Towards making block-based programming accessible for blind users. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. IEEE, 67–69.

[57] Stephanie Ludi, Jamie Simpson, and Wil Merchant. 2016. Exploration of the use of auditory cues in code comprehension and navigation for individuals with visual impairments in a visual programming environment. In *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility*. 279–280.

[58] Qianou Ma, Tongshuang Wu, and Kenneth Koedinger. 2023. Is AI the better programming partner? Human-Human pair programming vs. Human-AI pAIr programming. *arXiv preprint arXiv:2306.05153* (2023).

[59] Sean Mealin and Emerson Murphy-Hill. 2012. An exploratory study of blind software developers. In *2012 ieee symposium on visual languages and human-centric computing (vl/hcc)*. IEEE, 71–74.

[60] Matthew B Miles. 1994. Qualitative data analysis: An expanded sourcebook. *Thousand Oaks* (1994).

[61] Meredith Ringel Morris, Jazette Johnson, Cynthia L Bennett, and Edward Cutrell. 2018. Rich representations of visual content for screen reader users. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–11.

[62] Aboubakar Mountapmbeme, Obianuju Okafor, and Stephanie Ludi. 2022. Accessible blockly: An accessible block-based programming library for people with visual impairments. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility*. 1–15.

[63] Aboubakar Mountapmbeme, Obianuju Okafor, and Stephanie Ludi. 2022. Addressing accessibility barriers in programming for people with visual impairments: A literature review. *ACM Transactions on Accessible Computing (TACCESS)* 15, 1 (2022), 1–26.

[64] Omar Moured, Morris Baumgarten-Egemole, Karin Müller, Alina Roitberg, Thorsten Schwarz, and Rainer Stiefelhagen. 2024. Chart4blind: An intelligent interface for chart accessibility conversion. In *Proceedings of the 29th International Conference on Intelligent User Interfaces*. 504–514.

[65] Peya Mowar, Yi-Hao Peng, Aaron Steinfeld, and Jeffrey P Bigham. 2024. Tab to Autocomplete: The Effects of AI Coding Assistants on Web Accessibility. In *Proceedings of the 26th International ACM SIGACCESS Conference on Computers and Accessibility*. 1–6.

[66] 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 CHI Conference on Human Factors in Computing Systems*. 1–16.

[67] 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 CHI Conference on Human Factors in Computing Systems*. 1–26.

[68] Jakob Nielsen. 2023. AI: First New UI Paradigm in 60 Years. *Nielsen Norman Group* 18, 06 (2023), 2023.

[69] Sadia Nowrin, Patricia Ordóñez, and Keith Vertanen. 2022. Exploring Motor-impaired Programmers' Use of Speech Recognition. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility*.

[70] Shakked Noy and Whitney Zhang. 2023. Experimental evidence on the productivity effects of generative artificial intelligence. *Science* 381, 6654 (2023), 187–192.

[71] Sushil K Oswal and Hitender K Oswal. 2024. Examining the accessibility of generative AI website builder tools for blind and low vision users: 21 best practices for designers and developers. In *2024 IEEE International Professional Communication Conference (ProComm)*. IEEE, 121–128.

[72] Linus Påhlstorp and Lukas Gwardak. 2007. Exploring usability guidelines for rich internet applications. (2007).

[73] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590* (2023).

[74] Helen Petrie and Omar Kheir. 2007. The relationship between accessibility and usability of websites. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 397–406.

[75] Mahika Phutane, Crescentia Jung, Niu Chen, and Shiri Azenkot. 2023. Speaking with My Screen Reader: Using Audio Fictions to Explore Conversational Access to Interfaces. In *Proceedings of the 25th International ACM SIGACCESS Conference on Computers and Accessibility*. 1–18.

[76] Augustin Popa. 2023. *Using Copilot Chat with C++ in Visual Studio: Generate Code, Fix Functions, and More*. https://devblogs.microsoft.com/visualstudio/using-copilot-chat-with-c-in-visual-studio-generate-code-fix-functions-and-more/ Microsoft Visual Studio Blog.

[77] Venkatesh Potluri, Maulishree Pandey, Andrew Begel, Michael Barnett, and Scott Reitherman. 2022. Codewalk: Facilitating shared awareness in mixed-ability collaborative software development. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility*. 1–16.

[78] Venkatesh Potluri, Priyan Vaithilingam, Suresh Iyengar, Y Vidya, Manohar Swaminathan, and Gopal Srinivasa. 2018. Codetalk: Improving programming environment accessibility for visually impaired developers. In *Proceedings of the 2018 chi conference on human factors in computing systems*. 1–11.

[79] Mercy Rajaselvi, F Jane Gloria, V Mohitha, and Guhan Selvarajan. 2021. A survey of programming editors for the visually impaired. *Accessed: Aug* 12 (2021).

[80] Sharon M Ravitch and Matthew Riggan. 2016. *Reason & rigor: How conceptual frameworks guide research*. Sage Publications.

[81] Joshua Robins. 2019. *Barrier Determination Framework for Video Game Analysis Regarding Users with Visual Impairments*. Ph. D. Dissertation. University of Huddersfield.

[82] Hyman Rodman. 1980. Are conceptual frameworks necessary for theory building? The case of family sociology. *Sociological Quarterly* 21, 3 (1980), 429–441.

[83] Lucas Rosenblatt, Patrick Carrington, Kotaro Hara, and Jeffrey P. Bigham. 2018. Vocal Programming for People with Upper-Body Motor Impairments. In *Proceedings of the 15th International Web for All Conference* (Lyon, France) *(W4A '18)*. Association for Computing Machinery, New York, NY, USA, Article 30, 10 pages. https://doi.org/10.1145/3192714.3192821

[84] Daniel Russo. 2024. Navigating the complexity of generative ai adoption in software engineering. *ACM Transactions on Software Engineering and Methodology* (2024).

[85] Saiph Savage, Claudia Flores-Saviaga, Rachel Rodney, Liliana Savage, Jon Schull, and Jennifer Mankoff. 2022. The global care ecosystems of 3D printed assistive devices. *ACM Transactions on Accessible Computing* 15, 4 (2022), 1–29.

[86] Ather Sharif, Sanjana Shivani Chintalapati, Jacob O Wobbrock, and Katharina Reinecke. 2021. Understanding screen-reader users' experiences with online data visualizations. In *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. 1–16.

[87] Thomas B Sheridan. 1992. *Telerobotics, automation, and human supervisory control*. MIT press.

[88] Patricia M Shields and Nandhini Rangarajan. 2013. *A playbook for research methods: Integrating conceptual frameworks and project management*. New Forums Press.

[89] Kristen Shinohara, Murtaza Tamjeed, Michael McQuaid, and Dymen A. Barkins. 2022. Usability, Accessibility and Social Entanglements in Advanced Tool Use by Vision Impaired Graduate Students. *Proc. ACM Hum.-Comput. Interact.* 6, CSCW2, Article 551 (nov 2022), 21 pages. https://doi.org/10.1145/3555609

[90] Ann C Smith, Justin S Cook, Joan M Francioni, Asif Hossain, Mohd Anwar, and M Fayezur Rahman. 2003. Nonvisual tool for navigating hierarchical structures. *ACM SIGACCESS Accessibility and Computing* 77-78 (2003), 133–139.

[91] Mads Soegaard and Rikke Friis Dam. 2012. The encyclopedia of human-computer interaction. *The encyclopedia of human-computer interaction* (2012).
[92] Stack Overflow. 2022. *Stack Overflow Developer Survey 2022.* https://survey.stackoverflow.co/2022/ Accessed: 2023-12-06.
[93] Stack Overflow. 2023. *Stack Overflow Developer Survey 2023.* https://visualstudiomagazine.com/articles/2023/06/28/so-2023.aspx Accessed: 2023-12-06.
[94] Andreas Stefik, Roger Alexander, Robert Patterson, and Jonathan Brown. 2007. WAD: A feasibility study using the wicked audio debugger. In *15th IEEE International Conference on Program Comprehension (ICPC'07).* 69–80.
[95] Andreas M Stefik, Christopher Hundhausen, and Derrick Smith. 2011. On the design of an educational infrastructure for the blind and visually impaired in computer science. In *Proceedings of the 42nd ACM technical symposium on Computer science education.* 571–576.
[96] Sarit Felicia Anais Szpiro, Shafeka Hashash, Yuhang Zhao, and Shiri Azenkot. 2016. How people with low vision access computing devices: Understanding challenges and opportunities. In *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility.* 171–180.
[97] Delphine Szymczak. 2023. Tools in and out of sight: an analysis informed by Cultural-Historical Activity Theory of audio-haptic activities involving people with visual impairments supported by technology. (2023).
[98] A Tlili et al. 2021. Game-based learning for learners with disabilities–what is next? A systematic literature review from the activity theory perspective. Front. Psychol. 12, 814691 (2022).
[99] Emmanuel Utreras and Enrico Pontelli. 2020. Accessibility of block-based introductory programming languages and a tangible programming tool prototype. In *International Conference on Computers Helping People with Special Needs.* Springer, 27–34.
[100] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts.* 1–7.
[101] Visual Studio Code Documentation. 2023. *GitHub Copilot Overview: Seamless Integration with Visual Studio Code.* https://code.visualstudio.com/docs/copilot/overview Accessed: 2023-12-06.
[102] Lev S Vygotsky. 2012. *Thought and language.* MIT press.
[103] Ruotong Wang, Ruijia Cheng, Denae Ford, and Thomas Zimmermann. 2024. Investigating and designing for trust in ai-powered code generation tools. In *The 2024 ACM Conference on Fairness, Accountability, and Transparency.*
[104] Justin D Weisz, Jessica He, Michael Muller, Gabriela Hoefer, Rachel Miles, and Werner Geyer. 2024. Design Principles for Generative AI Applications. In *Proceedings of the CHI Conference on Human Factors in Computing Systems.*
[105] Man-Fai Wong, Shangxin Guo, Ching-Nam Hang, Siu-Wai Ho, and Chee-Wei Tan. 2023. Natural language generation and understanding of big code for AI-assisted programming: A review. *Entropy* 25, 6 (2023), 888.
[106] Jason Yu and Cheryl Qi. 2024. The Impact of Generative AI on Employment and Labor Productivity. *Review of Business* 44, 1 (2024).
[107] Lotus Zhang, Simon Sun, and Leah Findlater. 2023. Understanding Digital Content Creation Needs of Blind and Low Vision People. In *Proceedings of the 25th International ACM SIGACCESS Conference on Computers and Accessibility.*
[108] Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming.* 21–29.
[109] Jonathan Zong, Crystal Lee, Alan Lundgard, JiWoong Jang, Daniel Hajas, and Arvind Satyanarayan. 2022. Rich screen reader experiences for accessible data visualization. In *Computer Graphics Forum*, Vol. 41. Wiley Online Library.

A Study Questions

Participant Background & Copilot Experience

- Can you tell us how long you have been working as a professional software developer?
- Which programming language do you use most frequently in your projects?
- How long have you been using `<programming language>` as your main programming language?
- Have you ever used something like Copilot or ChatGPT for coding?
- Can you tell me about a time when you used a tool like Copilot for coding? What worked well and what didn't?
  - Can you share a time when it helped?
  - Can you share a time when it did not help?

Training session

- Participants took part in a training session where they were introduced to GitHub Copilot and its features. The training also covered how Copilot integrates with accessible development environments and assistive technologies that participants were already using. This session aimed to ensure that all participants started with a fundamental understanding of how to interact with Copilot during programming tasks.

Hands-on coding & Debugging task

Each participant was given a moderately complex programming task to complete using GitHub Copilot in Visual Studio Code, as well as a related debugging task, also to be completed using GitHub Copilot with Visual Studio Code.

- **Programming Task**. Based on prior work [33], the programming task we selected for participants was to write a program that takes a string representing the user's birthday, using their preferred programming language. The string the user provides is in either the format DDMMYY for day, month, and year, or DDMMYYYY. The program takes the user's birthday and should output the number of days until that person's next birthday. We asked participants to ensure their solution contains a class that implements all of the date functionality (e.g., it should have a constructor that takes in the string). We also asked them to write unit tests for the class they created. Participants were asked to "think aloud," verbalizing their thought process as they interacted with Copilot and wrote code. This provided insights into real-time decision-making and interaction patterns.
- **Debugging Task**. Following the initial coding task, a debugging session was conducted where participants needed to identify and correct any errors in the code generated by Copilot. During this phase, participants were also asked about various aspects, including their usual approach for evaluating code accuracy, how they dealt with encountered issues, and their thoughts on whether Copilot made their debugging process simpler or more complex.

Post-Study Questions

- I am interested in how you organize code in your head. When you are reading a lot of new code, what is your approach?
- When you are writing new code or modifying existing code, how do you plan out what to do in your head or organize your ideas?
- How do you usually read through or navigate your code? How did Copilot impact this process?
- Does Copilot change how you do this? How?
- Did Copilot suggestions change your plan at all?
- What was the experience of using Copilot like for you?
  - Did it help with completing the task? Why or why not?
  - What would make Copilot better for you?
  - What was the most difficult thing about using Copilot?
- Now that you have used Copilot, what is your feel for the scenarios where you would use it and not use it?

- What are your personal concerns regarding using AI in your work?