AI-Assisted Contextual Code Narration for the Visually Impaired*

Prajwal Narayanaswamy[1], Abbas Attarwala[1], Paul Viotti[2],
Jaime Raigoza[1] and Ed Lindoo[3]
[1]Computer Science Department
California State University, Chico
Chico, CA, 95973
plnu@csuchico.edu, aattarwala@csuchico.edu, jraigoza@csuchico.edu
[2]Political Science Department
California State University, Chico
Chico, CA, 95973
pviotti@csuchico.edu
[3]Computer Information Systems
Anderson School of Business
Regis University
Denver, CO, 80221
elindoo@regis.edu

Abstract

This paper introduces a new methodology designed to enhance the accessibility of source code for computer programmers who are blind or visually impaired (BVI). Conventional screen readers, such as JAWS, typically interpret and vocalize code in a linear, word-by-word fashion, often failing to convey the contextual relationships and structural hierarchy fundamental to program comprehension. To overcome these limitations, we propose a system that leverages the OpenAI large language model (LLM), guided by carefully engineered prompts, to transform source code

*Copyright ©2025 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

into structured, context-aware natural language descriptions. For example, rather than sequentially narrating deeply nested conditional statements line by line, our system first summarizes the structural context beforehand (e.g., identifying a nested conditional statement five levels deep) and subsequently details each condition clearly within this context. Our system, implemented as a Visual Studio Code extension, uses Eleven Labs Text-to-Speech (TTS) to read code aloud with two distinct voices. We designate a female voice, `Rachel`, to read each line of source code verbatim and a male voice, `Andy`, to provide contextual explanations. Although this approach is designed to significantly enhance BVI' comprehension of code structure and logic, providing a more intuitive and efficient auditory coding experience, it has not yet been formally evaluated with BVI individuals.

1 Introduction

Programming and software development are increasingly essential skills in numerous fields. However, BVI developers often encounter significant barriers due to the inherent limitations of traditional screen reader software, which typically vocalizes code in a linear, line-by-line fashion [9, 2]. This approach, although functional for basic textual navigation, struggles to convey the structural and logical context critical for comprehending complex programming constructs, such as deeply nested conditional statements and iterative loops.

To address these challenges, we propose a solution that takes advantage of recent advances in artificial intelligence (AI), with a particular emphasis on LLMs such as those developed by OpenAI, to improve the auditory rendering of source code. By employing meticulously designed prompt engineering techniques, our approach enables LLMs to generate structured, context-aware summaries of code that substantially enhance its comprehensibility when delivered through TTS systems. For example, when processing complex constructs, such as nested loops or conditional statements, the system initially conveys a high-level overview of the control structure, followed by a detailed narration of individual components. This method preserves contextual coherence and significantly improves the user's ability to cognitively parse and navigate programming logic in an auditory format. To accommodate diverse user preferences and usage scenarios, the proposed solution also offers an alternative mode in which BVI users can access the source code in a verbatim, line-by-line format without contextual augmentation. This functionality ensures greater flexibility, allowing BVI users to select between enriched contextual summaries and a direct, unmodified auditory representation of the code based on their individual needs or task requirements. Implemented as an extension within Visual Studio Code (VSCode), a widely adopted Integrated Development En-

vironment (IDE), this tool integrates seamlessly into the existing workflows of developers, promoting both accessibility and ease of adoption. By bridging the gap between traditional screen readers and meaningful code comprehension, our approach empowers BVI programmers to engage more fully and efficiently in software development activities.

2 Literature Review

BVI individuals remain severely underrepresented in software development. In a large developer survey, only 2% of respondents reported having a disability of some kind, including visual impairments [9]. Researchers have identified significant accessibility barriers in current programming tools as a contributing factor to this underrepresentation [9].

Contemporary code editors and IDEs rely heavily on visual cues such as syntax highlighting, indentation, and graphical debuggers, which screen readers struggle to capture or interpret effectively. Potluri et al. [11] categorize the accessibility challenges of IDEs into four areas: *discoverability*, *glanceability*, *navigability*, and *alertability*. Glanceability refers to the ability to quickly skim and grasp code structure at a glance—something sighted developers achieve through visual scanning, while screen reader users must process information linearly. For example, sighted developers can quickly navigate through code using scroll, point, and click, while screen reader users have limited navigation options [11]. Subsequent research has confirmed that these challenges persist in modern development environments [6].

A recent systematic review of the literature by [9] confirms these persistent barriers, from code navigation and understanding of the state of the program to the lack of accessible debugging tools, that continue to hinder BVI students and developers. Without additional support, BVI programmers must expend excessive time and cognitive effort to comprehend others' code or keep track of program context, which in turn deters many from pursuing coding as a career or academic interest.

Efforts to address these challenges have focused on adapting or augmenting the development environment for non-visual interaction. [11] introduced *CodeTalk*, a Visual Studio plugin that provides auditory cues and structured code navigation for BVI developers. CodeTalk addresses glanceability challenges, specifically the difficulty of quickly obtaining overview information when navigating code linearly with a screen reader, by providing keyboard shortcuts that generate accessible tree views of code structure (e.g., namespaces, classes, and functions) and accessible lists of specific code elements. The plugin also announces IDE events (such as real-time compiler errors or debugging information) through audio cues that would otherwise go unnoticed by a screen reader

user.

Another approach has been to redesign the editing paradigm itself: Grid Editor [6] is a recent system that represents source code in a two-dimensional spreadsheet-like grid, with each row and column denoting code lines and nesting levels, respectively. This structured layout, coupled with custom keyboard navigation and audio cues, enables BVI programmers to navigate and edit code more efficiently and accurately than in a plain text editor.

Likewise, researchers have explored making block-based visual programming languages usable through non-visual means. [16] replaced the complex spatial navigation required by block-based environments such as Blockly with a simple line-by-line navigation interface and reported that this significantly reduced task completion times and improved usability ratings in an evaluation with blindfolded blind participants. Despite these advances, many solutions require specialized hardware or custom environments (e.g., tactile displays, bespoke editors), and thus have seen limited adoption in mainstream programming practice [9]. The need remains for integrated accessibility features that can be easily deployed in popular coding tools to bridge the gap between sighted and visually impaired programmers.

The emergence of AI and LLMs offers new possibilities to improve programming accessibility. LLM-based assistants can generate natural-language descriptions of code behavior, summarize complex code sections, and answer clarifying questions about code, effectively acting as on-demand narrators. Recent research has begun to apply these capabilities to assist BVI developers. For example, [12] present *BLVRUN*, a command-line tool that intercepts runtime error traces and uses an LLM to generate concise, informative summaries of Python traceback errors. This helps BVI programmers debug faster by cutting through verbose error logs and highlighting the relevant causes of an error in plain language. In the context of education, [16] integrated a conversational AI (based on a code-capable LLM) into their accessible block-language environment to produce high-level summaries of the program's purpose. Their user study with blindfolded sighted participants found that those who received these AI-generated code summaries reported high levels of comprehension and reduced anxiety in subjective evaluations, although testing with actual blind users remains future work.

These AI-driven approaches represent a significant advancement over earlier accessibility solutions, such as CodeTalk's TalkPoints, which relied on predefined speech and non-speech audio cues for debugging [11]. Although CodeTalk's audio cues required manual configuration, contemporary LLM systems can dynamically generate contextual explanations without explicit setup. More broadly, LLM assistants such as GitHub Copilot and ChatGPT have been touted as equalizers in programming education. By offering immediate

access to explanations, contextual hints, and documentation retrieval, these tools offer particular promise to learners who encounter barriers to traditional programming resources, including individuals with disabilities [7]. In this context, [7] observed that BVI developers are optimistic about the potential of AI coding assistants to improve their workflow by automating repetitive tasks and providing timely guidance, thus improving their productivity and coding efficiency. However, these benefits are highly dependent on accessible design. If implemented incorrectly, AI-based tools might introduce new frustrations for BVI users. For example, the study highlights that GitHub Copilot occasionally inundates users with excessive suggestions and creates difficulties in managing focus between the code editor and the assistant's output. Participants in that study emphasized the need for "AI timeouts" or controls to prevent AI from flooding the screen reader with information.

A recent study by [1] examined how BVI people use general-purpose AI chatbots such as ChatGPT in everyday tasks. Through qualitative interviews, the authors found that while BVI users frequently rely on these tools to gain information or solve problems (including coding questions), they often have to devise workarounds to navigate chatbot interfaces with a screen reader and must carefully verify AI outputs for accuracy. These findings highlight that the mere inclusion of AI assistance does not inherently guaranty accessibility. Instead, any AI-driven accessibility tool should strive to deliver support in a controlled, contextual manner that aligns with the user's normal coding habits, rather than disrupting them.

Our approach draws upon established insights from accessibility research while introducing a fundamental shift in how code narration is generated and delivered. CodeTalk [11] showed that BVI developers benefit from proactive information extraction, as screen reader users "must actively seek out information from various components of the IDE." CodeTalk provided accessible tree views and audio debugging cues (TalkPoints), but these required manual configuration by developers. Grid Editor [6] demonstrated the value of alternative code representations through coordinated text and grid views, though this required a dedicated environment separate from standard editors. Similarly, Stefik et al. [15] validated that auditory cues representing lexical scoping can improve program comprehension, using symbolic sounds to convey code structure.

Building on these foundations, our VS Code extension introduces two key innovations. First, rather than requiring manual configuration like CodeTalk's TalkPoints or using symbolic sounds like Stefik's scoping cues, we leverage OpenAI's LLM to automatically generate natural-language explanations of code structure and purpose in real-time. Second, we employ a dual-voice design for perceptual separation: a female voice reads the code verbatim, while a male

voice provides a contextual explanation. This separation reduces cognitive load when processing complex code, addressing the glanceability challenges identified in prior work. Importantly, our extension operates seamlessly within the standard VS Code interface and activates on-demand during navigation, avoiding both the workflow disruption of dedicated environments and the "constant deluge of suggestions" observed by [7].

The need for such capabilities is underscored by CodeTalk's user study, where one participant noted: "I never knew how much information I was not getting because I was using a screen reader" [11]. This remark highlights how accessibility barriers can become normalized, emphasizing the importance of systems that actively reveal the structural and semantic aspects of code that sighted developers can access visually.

3 Implementation Overview

Our accessibility solution is implemented as a VS Code extension. This choice is driven by the editor's widespread adoption among developers, its ease of integration, and its flexible architecture. According to the Stack Overflow 2024 Developer Survey [14], VS Code is used by 74% of developers, making it the most popular IDE by a significant margin—more than double the usage of its nearest alternative, Visual Studio (29%). While other indices such as the PYPL Top IDE Index [4] rank Visual Studio higher in terms of tutorial search popularity (28.48% versus VS Code's 15.27%), they also highlight the rapid growth trend of VS Code over the past five years. This widespread usage ensures that an extension developed for VS Code has the potential to reach a large and diverse audience of developers.

Beyond popularity, VS Code offers architectural advantages that make it suitable for this project [17]. It provides a well-documented extension API that enables integration of custom features. This extensibility allows the extension to automatically process code upon file opening and trigger audio playback when users interact with specific lines. VS Code also supports a wide range of programming languages, including C++, which remains relevant in many domains. Targeting VS Code allows us to embed the extension in a familiar and widely used environment for C++ developers. Its cross-platform availability on Windows, macOS, and Linux further enhances accessibility, while its large and active community contributes valuable support and resources [18].

The core functionality of the extension involves processing C++ source code and generating audio representations. When a user opens a C++ file, the extension parses it line-by-line. For each line, two audio outputs are generated: a verbatim narration of the code spoken in a female voice (which we refer to as `Rachel`) and a contextual explanation of the purpose of the line delivered in

a male voice (which we refer to as `Andy`). The contextual explanation is created by sending each line, along with a carefully crafted prompt, to the OpenAI API. This uses the capabilities of LLMs, which are trained in an extensive corpus of text and code. The prompt is explicitly designed to guide the model to produce explanations that are context-aware, aiming to describe not just what the line does but also why it exists in the broader logic of the program.

Once both verbatim narration and contextual explanation are generated, the system uses Eleven Labs TTS API to convert them into audio. For each line of code i, two separate audio files are created: `rachel_i.mp3` for verbal narration and `andy_i.mp3` for contextual explanation. Eleven Labs was selected for its high-quality, lifelike voice synthesis, its support for voice customization, and its flexibility in terms of language and integration options. The platform allows developers to choose from a variety of voices or even clone specific ones, which enables the clear separation of narration types via distinct male and female voices. In addition, Eleven Labs actively promotes its use for accessibility-focused applications, which aligns well with the goals of this paper. The API also supports customization of speech parameters, such as speed and stability, which offers room for further tuning of the user experience, which we plan to explore in future work of this extension.

A significant technical challenge in this project was achieving reliable sequential playback of the two audio narrations for each line of code. Ensuring that `Andy`' contextual explanation consistently follows `Rachel`' verbal narration—without gaps or overlap—proved difficult due to the lack of reliable playback completion triggers in the development environment. To address this, we used the `ffmpeg` multimedia framework to concatenate the two separate audio files into a single synchronized file `output_i.mp3` for each line i. By combining both narration types into a single file, the extension simplifies playback logic: when a user selects a line of code, only one audio file needs to be played. By default, this file contains `Rachel`'verbatim narration followed by `Andy`' contextual explanation. However, BVI users can choose to hear only `Rachel` or `Andy`, depending on their preference. This guaranties smooth and immediate sequential playback, resulting in a better and more consistent user experience.

4 Rationale and Hypothesized Benefits

We hypothesize that employing two distinct voices: 1) one designated for reading verbatim code syntax and 2) for providing contextual explanations can significantly enhance code comprehension for BVI users compared to the conventional approach of using a single TTS voice for all output.

The underlying rationale for this hypothesis is based on principles of auditory scene analysis, which involves parsing and grouping sounds to form

representations of distinct auditory objects [13]. By assigning different voices to the code and the explanation, the aim is to leverage this natural perceptual mechanism to help listeners more easily differentiate these two critical streams of information, potentially reducing ambiguity and cognitive effort. Our hypothesis is that the use of distinct female (verbatim code) and male (contextual explanation) voices improves auditory code comprehension for BVI users. This approach takes advantage of auditory scene analysis principles, where voice differences in pitch and timbre facilitate perceptual stream segregation [3, 8]. By perceptually separating the code syntax from its semantic explanation using distinct voices (male and female), our aim is to reduce the cognitive load associated with simultaneously processing syntax and semantics, similar to techniques used to separate real-world and virtual sounds in mixed-reality environments [5] Therefore, while the hypothesis is plausible and grounded in sound perceptual principles, its practical efficacy to improve code comprehension remains unproven. The potential cognitive benefits of easier stream segregation must be weighed against the potential costs of attention management, and this balance can only be determined through direct empirical testing on realistic coding tasks. In future studies, we hope to perform a randomized control trial experiment and validate the efficacy of our extension in VSCode.

5 Results

Table 1 presents a line-by-line comparison of how the selection sort algorithm, shown in Figure 1, is transcribed and narrated by three different systems: JAWS (based on our limited testing with the trial version), the verbatim `Rachel` voice, and the contextual Andy voice, both generated using ElevenLabs' TTS API. The actual audio recordings corresponding to Table 1, including playback of the voices of JAWS and `Rachel` and `Andy` , are available in an audio-only format [10]. We found that character-by-character playback from the JAWS trial version posed significant challenges for understanding code structure and logic, although we acknowledge that other configurations may be available in the full version.

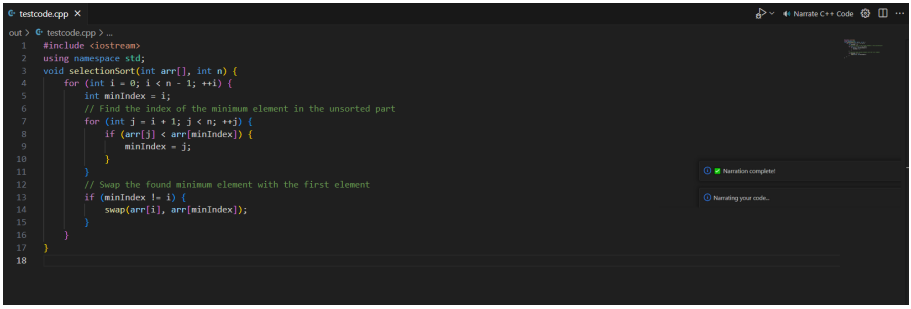

Figure 1: Selection Sort algorithm displayed in VSCode. The top-right corner shows the `Narrate C++ Code` button, which triggers audio narration using `Rachel` and `Andy` voices. Once narration is complete for all code lines, a pop-up appears in the bottom-right corner with a green checkbox labeled `Narration Complete`.

While the `Rachel` voice provides a clear, line-by-line reading of the code syntax (e.g., "Line 3: void selectionSort open parenthesis..."), which is already an improvement over the character-by-character output observed from the JAWS trial version (e.g., "v o i d space s e l e c t i o n S o r t..."), the `Andy` voice demonstrates a deeper level of understanding.

The key advantage lies in the OpenAI-powered contextual narration provided by `Andy`, which synthesizes a high-level explanation of each code line rather than simply reading it aloud. This approach leverages the LLM capabilities of OpenAI to deliver concise summaries that preserve the programmer's intent As highlighted in Line 7, `Andy` does not read the line of code verbatim but mentions that Line 7 is an inner loop inside an outer loop and that the inner loop will iterate over the unsorted part of the array starting from $i+1$. This semantic interpretation, derived from processing the code with an OpenAI model prompted for context-aware explanations, is something traditional screen readers typically cannot provide.

Further examples of this contextual understanding are evident throughout the table: Line 4: `Andy` explains the purpose of the outer loop within the selection sort algorithm ("placing the correct element at position i") and the reasoning behind the $n-1$ condition. Line 7: `Andy` clarifies the role of the inner loop ("examining every element in the unsorted portion") and its starting point relative to the outer loop ($i+1$). Lines 8 & 9: `Andy` explains the conditional logic being performed (checking for a smaller element) and the action taken if the condition is met (updating $minIndex$). Lines 15, 16, & 17: Each of these lines is in a closed curly bracket. `Andy` explains clearly that the first curly bracket closes the the if block. The second curly bracket closes the outer loop

and the last curly bracket closes the selection sort function.

This demonstrates that the `Andy` voice, powered by OpenAI, captures and conveys the semantic meaning and structural context of the code, offering a significantly richer and more comprehensible auditory experience for users compared to the literal, non-interpretive playback characteristic of standard screen readers like the JAWS version tested.

Line #	Code Line	JAWS Transcription	Rachel Voice Transcription	Andy Voice Transcription
Line 1	`#include <iostream>`	`# i n c l u d e < i o s t r e a m >`	Line 1: #include less than iostream greater than	Line 1: This line includes the inputoutput stream library for Cplusplus.
Line 2	`using namespace std;`	`u s i n g n a m e s p a c e s t d ;`	Line 2: using namespace std semicolon	Line 2: Here, the code is indicating the usage of the standard namespace in Cplusplus.
Line 3	`void selectionSort(int arr[], int n) {`	`v o i d s e l e c t i o n S o r t ( i n t a r r [ ] , i n t n ) {`	Line 3: void selectionSort open parenthesis int arr open square bracket close square bracket int n close parenthesis open curly bracket	Line 3: This line defines a function named selectionSort that takes an integer array and its size as parameters.
Line 4	`for (int i = 0; i < n - 1; ++i) {`	`f o r ( i n t i = 0 ; i < n - 1 ; + + i ) {`	Line 4: for open parenthesis int i = 0 semicolon i less than n minus 1 semicolon plusplusi close parenthesis open curly bracket	Line 4: Inside the selectionSort function, there is a for loop that iterates over the array. This loop will run n minus 1 times.
Line 5	`int minIndex = i;`	`i n t m i n I n d e x = i ;`	Line 5: int minIndex = i semicolon	Line 5: Within the loop, an integer variable minIndex is declared and initialized with the value of i.
Line 6	`// Find the index of the minimum element in the unsorted part`	`/ / F i n d t h e i n d e x o f t h e m i n i m u m e l e m e n t i n t h e u n s o r t e d p a r t`	Line 6: comment Find the index of the minimum element in the unsorted part	Line 6: This is a comment describing the purpose of finding the index of the minimum element in the unsorted part.
Line 7	`for (int j = i + 1; j < n; ++j) {`	`f o r ( i n t j = i + 1 ; j < n ; + + j ) {`	Line 7: for open parenthesis int j = i plus 1 semicolon j less than n semicolon plusplusj close parenthesis open curly bracket	Line 7: Inside the outer loop, there is another nested loop that iterates over the unsorted part of the array starting from iplus1.
Line 8	`if (arr[j] < arr[minIndex]) {`	`i f ( a r r [ j ] < a r r [ m i n I n d e x ] ) {`	Line 8: if open parenthesis arr open square bracket j close square bracket less than arr open square bracket minIndex close square bracket close parenthesis open curly bracket	Line 8: This line checks if the element at index j is less than the element at the current minimum index to update the minimum index if needed.
Line 9	`minIndex = j;`	`m i n I n d e x = j ;`	Line 9: minIndex = j semicolon	Line 9: If the condition in line 8 is true, minIndex is updated to the current index j.
Line 10	`}`	`}`	Line 10: close curly bracket	Line 10: This is the end of the inner loop where the comparison is made to find the minimum element.
Line 11	`}`	`}`	Line 11: close curly bracket	Line 11: End of the inner loop block.
Line 12	`// Swap the found minimum element with the first element`	`/ / S w a p t h e f o u n d m i n i m u m e l e m e n t w i t h t h e f i r s t e l e m e n t`	Line 12: comment Swap the found minimum element with the first element	Line 12: This comment indicates the intention to swap the found minimum element with the element at the first index after the inner loop completes all iterations
Line 13	`if (minIndex != i) {`	`i f ( m i n I n d e x ! = i ) {`	Line 13: if open parenthesis minIndex negation = i close parenthesis open curly bracket	Line 13: Here, an if condition checks if the minIndex is not equal to i, meaning a smaller element was found, then it proceeds to swap the elements.
Line 14	`swap(arr[i], arr[minIndex]);`	`s w a p ( a r r [ i ] , a r r [ m i n I n d e x ] ) ;`	Line 14: swap open parenthesis arr open square bracket i close square bracket, arr open square bracket minIndex close square bracket close parenthesis semicolon	Line 14: Within the if condition, the swap function is called to interchange the elements at index i and minIndex.
Line 15	`}`	`}`	Line 15: close curly bracket	Line 15: End of the condition block checking for the swap condition.
Line 16	`}`	`}`	Line 16: close curly bracket	Line 16: End of the outer loop block that iterates through the array n minus 1 times to sort the elements.
Line 17	`}`	`}`	Line 17: close curly bracket	Line 17: End of the selectionSort function block.

Table 1: Comparison of Code Transcriptions by JAWS, Female Voice, and Male Voice (Lines 1–17) on the selection sort algorithm.

6 Conclusion

This paper presented an accessibility extension designed to enhance the comprehension of C++ code for BVI developers within the widely used VSCode environment. The system employs two distinct synthetic voices generated via ElevenLabs: a female voice (`Rachel`) provides a precise, verbatim reading of each code line, ensuring syntactic accuracy, while a male voice (`Andy`) delivers

a context-aware explanation. Contextual narration is generated using the OpenAI API, which analyzes each line of code to infer semantic meaning and describe its role within the broader program structure. This dual-voice approach aims to offer a deeper level of understanding than traditional screen readers, which typically provide only literal readings. The use of distinct voices also supports auditory stream segregation, potentially reducing the cognitive load on the listener. The extension integrates inside VSCode, allowing BVI users to navigate code using the up and down arrow keys and trigger the synchronized playback of both the verbatim and contextual narrations for the selected line on demand. Together, these features form a cohesive framework for auditory code comprehension that bridges the gap between syntactic precision and semantic understanding. Future work will involve conducting user studies to evaluate the effectiveness of the tool in improving comprehension and efficiency, as well as exploring extensions to additional programming languages.

Acknowledgment

We gratefully acknowledge the use of OpenAI's ChatGPT for proofreading, grammatical checks, and other text editing tasks.

References

[1] Rafi Adnin and Maitraye Das. "\"I look at it as the king of knowledge\": How Blind People Use and Understand Generative AI Tools". In: *Proceedings of the 26th ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '24)*. 2024. DOI: 10.1145/3663548.3675631.

[2] Khaled Albusays and Stephanie Ludi. "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*. 2016, pp. 82–85.

[3] Christian Brodbeck and Jonathan Z Simon. "Cortical tracking of voice pitch in the presence of multiple speakers depends on selective attention". In: *Frontiers in Neuroscience* 16 (2022), p. 828546.

[4] Pierre Carbonnelle. *Top IDE Index*. Accessed: 2025-04-19. 2023. URL: https://pypl.github.io/IDE.html.

[5] Ruei-Che Chang et al. "SoundShift: Exploring Sound Manipulations for Accessible Mixed-Reality Awareness". In: *Proceedings of the 2024 ACM Designing Interactive Systems Conference*. 2024, pp. 116–132.

[6] Ehtesham-Ul-Haque, Syed M. Monsur, and Shams M. Billah. "GridCoding: An Accessible, Efficient, and Structured Coding Paradigm for Blind and Low-Vision Programmers". In: *Proceedings of the 35th ACM Symposium on User Interface Software and Technology (UIST '22)*. ACM, 2022, 44:1–44:21. DOI: 10.1145/3526113.3545620.

[7] Carmen Flores-Saviaga et al. "The Impact of Generative AI Coding Assistants on Developers Who Are Visually Impaired". In: *arXiv preprint arXiv:2503.16491* (2025). URL: https://arxiv.org/abs/2503.16491.

[8] João Guerreiro and Daniel Gonçalves. "Faster text-to-speeches: Enhancing blind people's information scanning with faster concurrent speech". In: *Proceedings of the 17th international ACM SIGACCESS conference on computers & accessibility*. 2015, pp. 3–11.

[9] Aboubakar Mountapmbeme, Obianuju Okafor, and Stephanie Ludi. "Addressing accessibility barriers in programming for people with visual impairments: A literature review". In: *ACM Transactions on Accessible Computing (TACCESS)* 15.1 (2022), pp. 1–26.

[10] Prajwal Narayanaswamy. *Selection Sort Audio Demonstrations*. https://drive.google.com/drive/folders/1M63Xl-hAnTH4UpskM0dJEDGgcE1M9s2p?usp=share_link. Audio files comparing JAWS screen reader and Rachel and Andy narrations for a C++ selection sort implementation. 2025. URL: https://drive.google.com/drive/folders/1M63Xl-hAnTH4UpskM0dJEDGgcE1M9s2p?usp=share_link.

[11] Venkatesh Potluri et al. "CodeTalk: Improving Programming Environment Accessibility for Visually Impaired Developers". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018. DOI: 10.1145/3173574.3174192.

[12] Clark Saben, Jessica Zeitz, and Prashant Chandrasekar. "Enabling Blind and Low-Vision (BLV) Developers with LLM-Driven Code Debugging". In: *Journal of Computing Sciences in Colleges* 40.3 (2024), pp. 204–215.

[13] Joel S Snyder, Claude Alain, and Terence W Picton. "Effects of attention on neuroelectric correlates of auditory stream segregation". In: *Journal of cognitive neuroscience* 18.1 (2006), pp. 1–13.

[14] Stack Overflow. *2024 Developer Survey*. Accessed: 2025-04-19. 2024. URL: https://survey.stackoverflow.co/2024/.

[15] Andreas Stefik, Christopher Hundhausen, and Robert Patterson. "An empirical investigation into the design of auditory cues to enhance computer program comprehension". In: *International Journal of Human-Computer Studies* 69.12 (2011), pp. 820–838.

[16] Naoto Takamatsu et al. “Computer Programming Education System for Visually Impaired Individuals”. In: *Proceedings of the 14th International Workshop on Computer Science and Engineering (WCSE 2024)*. IEEE. 2024, pp. 64–69. DOI: 10.18178/wcse.2024.06.010.

[17] Sufyan Uzayr. *Mastering Visual Studio Code: A Beginner’s Guide*. CRC Press, 2022.

[18] Erin Yepis. *Developers want more, more, more: the 2024 results from Stack Overflow’s Annual Developer Survey*. Accessed: 2025-04-19. Jan. 2025. URL: https://stackoverflow.blog/2025/01/01/developers-want-more-more-more-the-2024-results-from-stack-overflow-s-annual-developer-survey/.