

Towards Automating Accessibility in Digital Authoring Workflows

Peya Mowar

CMU-RI-TR-25-61

July 14, 2025



The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:

Professor Jeffrey P. Bigham, *Co-Chair*
Professor Aaron Steinfeld, *Co-Chair*
Professor Deva Ramanan
Vivian Shen

*Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Robotics.*

Copyright © 2025 Peya Mowar. All rights reserved.

To my parents, for always leading by example.

Abstract

Most digital content today remains inaccessible to people with disabilities, who make up 16% of the global population. A long-standing challenge in accessible computing is ensuring digital authors consistently provide the metadata required to make their content accessible through assistive technologies. Despite numerous specialized accessibility tools, authors often lack the time, training, or incentive to use them effectively.

Recent advancements in AI bring an unprecedented opportunity for automating the production and preservation of accessibility metadata in the authoring process. This thesis argues that developers and content creators can use AI tools to make their digital content accessible with minimal disruption to their existing workflows.

It presents two novel AI authoring tools aimed at improving the digital accessibility of websites and scientific documents. The first, CodeA11y, is an AI coding assistant that helps frontend developers produce accessibility-compliant user interface code. The second, WYSIWYM tagging, preserves semantic metadata already present in authored source documents to generate tags and preserve reading order in PDFs.

Through comprehensive evaluations, this thesis demonstrates the effectiveness of these tools and represents a first step toward automating accessibility in mainstream digital authoring workflows.

Acknowledgments

I was told that pursuing a master’s program is like running a sprint. At CMU, it felt more like a relay. I am deeply grateful to have met incredible people here who helped carry this work forward. And I feel even more fortunate knowing they will continue cheering me on as I now take on the marathon that is the PhD.

First, I would like to thank my advisors, Jeff and Aaron, without whom the past two years would not have been nearly as enriching or enjoyable. From them, I learned how to effectively formulate and present my work, embrace responsibility, and ask for help when needed. They always believed in me and patiently guided me toward solving truly wicked problems. But perhaps what I cherished most was the worldly wisdom and fatherly advice that they generously shared in every meeting. I couldn’t have asked for better mentors!

Second, I am grateful to my committee members, Deva and Vivian, for their fresh perspectives and thoughtful feedback. I also appreciate the wonderful camaraderie of my fellow Big and TBD lab members - Abena, Amanda, Allan, Atieh, Avigyan, Chris, Faria, Hamza, Huy, Jason, Jessica, Kate, Kundan, Liz, Oscar, Quentin, Rayna, Sara, Suresh, Yasmine, and Yi-Hao. I was fortunate to help organize the CMU Accessibility Lunches, where conversations with Andy, Patrick, and others significantly shaped my work. I am indebted to all course instructors, MSR program staff, and the broader SCS community for fostering an environment that supported simultaneous growth and grounding.

Third, I was blessed to make many friends who quickly became like family away from home: Aadesh, Aman, Aryan, Avigyan, Dhruv, Ishita, Khush, Mehal, Neham, Parth, Rupali, Sagnik, Sashank, Shambhavi, Srujan, Tirtha, Vihaan, Vismitha, and Yash. They made a huge impact on my life and more unexpectedly, on my thesis, by participating in my user studies or spending hours on manual annotation. I am endlessly thankful to my roommate and silent co-author, Shivangi, who was always there to pull me out when I got stuck in the weeds. I could never have made it this far without any of you!

Fourth, I would be remiss not to thank Saikat and Mohit for teaching me the 101s of research and HCI, respectively, and for laying the foundation for everything that followed. To Ishani, Mini, and Ruchika – thanks for always having my back.


Finally, I thank my stars for a family whose unwavering love has always been my greatest strength. Thanks, Ishan, for carrying my joys and worries as your own. Thanks, Kuhu, for putting me on a pedestal I have yet to deserve. And my deepest gratitude to my parents, who have given me everything I have, taught me everything I know, and as my north stars, shown me all that is worth striving for.

Contents

| | | |
|----------|---|----------|
| 1 | Introduction | 1 |
| 1.1 | Document Overview | 2 |
| | On Producing Accessibility Metadata | 3 |
| 2 | CodeA11y: AI Coding Assistant for Accessible Web Development | 4 |
| 2.1 | Preamble | 4 |
| 2.2 | Introduction | 5 |
| 2.3 | Related Work | 6 |
| 2.3.1 | Assessing Web Accessibility | 6 |
| 2.3.2 | End-user Accessibility Repair | 8 |
| 2.3.3 | Developer Tools for Accessibility | 9 |
| 2.4 | Formative Study Methods | 10 |
| 2.4.1 | Tasks | 10 |
| 2.4.2 | Protocol | 11 |
| 2.4.3 | Participants | 12 |
| 2.4.4 | Procedure | 12 |
| 2.4.5 | Data Collection and Analysis | 13 |
| 2.5 | Formative Findings | 14 |
| 2.5.1 | Developer Behavior | 14 |
| 2.5.2 | AI Usage and Prompting Strategies | 15 |
| 2.5.3 | Implications for Web Accessibility | 16 |
| 2.6 | Design Requirements | 17 |
| 2.6.1 | (G1) Integrate System Prompts for Accessibility Awareness | 17 |
| 2.6.2 | (G2) Support Identification of Accessibility Issues | 17 |
| 2.6.3 | (G3) Encourage Developers to Complete AI-Generated Code | 18 |
| 2.7 | CodeA11y System Overview | 18 |
| 2.7.1 | Multi-Agent Architecture | 18 |
| 2.7.2 | User Interaction | 21 |
| 2.8 | User Evaluation | 22 |
| 2.8.1 | Methodology | 22 |
| 2.8.2 | Results | 24 |
| 2.9 | Discussion | 26 |
| 2.9.1 | Which comes first: Adoption or Awareness? | 27 |
| 2.9.2 | AI-Assisted but Developer-Completed | 27 |
| 2.9.3 | Limitations & Future Work | 28 |
| 2.10 | Conclusion | 30 |

| | |
|---|-----------|
| On Preserving Accessibility Metadata | 31 |
| 3 What You See Is What You Mean PDF Tagging | 32 |
| 3.1 Preamble | 32 |
| 3.2 Introduction | 33 |
| 3.3 Related Work | 34 |
| 3.3.1 Why Do Our Research Papers Remain Inaccessible? | 34 |
| 3.3.2 Can Vision-Language Models Help? | 35 |
| 3.4 Methodology | 36 |
| 3.4.1 Visual Rendering (PDF) Processing | 36 |
| 3.4.2 Source Document Parsing | 37 |
| 3.4.3 Aligning Visual and Source Representations | 38 |
| 3.4.4 Metadata Generation | 39 |
| 3.5 Experiments | 40 |
| 3.5.1 Dataset | 40 |
| 3.5.2 Metadata Quality Metrics | 40 |
| 3.5.3 Accessibility Criteria | 41 |
| 3.6 Results | 41 |
| 3.6.1 Quantitative Results | 41 |
| 3.6.2 Qualitative Results | 42 |
| 3.6.3 Limitations | 43 |
| 3.7 Conclusion | 45 |
| 4 Conclusion | 46 |
| 4.1 Key Contributions | 46 |

List of Figures

| | | |
|-----|---|----|
| 1.1 | Accessibility metadata () flows from (a) the author to (d) the consumer. The author produces metadata in (b) the authoring tool, which must preserve it in (c) the digital content. Assistive technologies then provide this metadata to the consumer. Chapters 2 and 3 detail tools that support the production and preservation of metadata respectively. | 2 |
| 2.1 | Examples of task descriptions and visual references given to our participants: (a) Task 2 was to implement a new contact form for subscribing to a mailing list, and (b) Task 3 was to add a ‘Top Stories’ section with linked articles. Successfully completing them required proper labeling of the form elements and links, but this was not explicitly stated in the instructions. | 11 |
| 2.2 | Mean Accessibility Evaluation Scores by Tasks and Copilot Usage: Higher scores indicate success. | 16 |
| 2.3 | CodeA11y is a GitHub Copilot Extension for Accessible Web Development. CodeA11y addresses accessibility limitations of Copilot observed in our study with developers through three features: (1) accessibility-by-default code suggestions, (2) automatic identification of relevant accessibility errors, and (3) reminders to replace placeholders in generated code. Integrating these features directly into AI coding assistants would improve the accessibility of the user interfaces (UIs) developers create. | 19 |
| 2.4 | CodeA11y Architecture: Multi-agent workflow | 21 |
| 2.5 | Contrasting responses for the same task across AI-assistants, showing differences in workflows. Developers had access to both the code and the rendered user interface. (a) and (d) represent conversations with the baseline assistant, and CodeA11y respectively. (b) and (e) show the buttons generated by each assistant in their default state. (c) and (f) display the buttons when hovered over, illustrating the differences in button color contrast. | 22 |
| 2.6 | Mean Accessibility Evaluation Scores by Tasks and AI Assistant: Higher scores indicate success. | 24 |

| | | |
|-----|---|----|
| 3.1 | Visualizations of metadata generated by (a) accessibility remediation tools (i, v), (b) machine learning pipelines for document layout analysis (ii–iv), (c) vision-language models for object detection (vi–viii), and (d) our tagging approach (ix). Existing approaches rely solely on visual features of the PDF, <i>i.e.</i> , adopting a “What You See Is What You Get” paradigm, producing cluttered, incomplete or incorrect metadata. In contrast, our “What You See Is What You Mean” tags are refined using additional context from source documents (e.g., LaTeX files), resulting in cleaner and more semantically accurate results. | 35 |
| 3.2 | Our proposed method comprises of four steps: (i) processing the visual rendering of PDFs (in blue background), (ii) parsing the source document (in gray background), (iii) aligning the visual and structural representations using an LLM, and (iv) generating the metadata (tags and reading order). | 37 |
| 3.3 | Classification error frequencies across 40 PDFs comprising 554 pages (N = 10,261 tags). Each bar represents the frequency of misclassification from a ground truth tag to a predicted tag (denoted as <True> to <Predicted>). The most common error involves classifying various tags as <Artifact>, indicated by <X> to <Artifact>. | 43 |
| 3.4 | Positive examples from our method: (a) Our approach accurately classifies figures, captions, and artifacts; (b) It correctly predicts footnote order while skipping artifacts. | 44 |
| 3.5 | Negative examples from our method: (a) Paragraphs in Japanese are tagged artifacts; (b) Figures are occasionally merged with captions or split into sub-figures. | 44 |

List of Tables

| | | |
|-----|--|----|
| 2.1 | Our formative study included four tasks. Each task was not primarily about accessibility but included an accessibility issue that was required to complete the task successfully. We adopt the scales of Unacceptable, Average and Good from prior work [68]. Uninformative attributes are those that merely reflect the field, such as ‘alt’ as alt-text or ‘click here’ as link description, without providing more meaningful or descriptive content [95]. Tasks are ranked from easy to difficult based on the time taken and success rates observed in our pilot studies. | 10 |
| 2.2 | Participant User Groups: Each group is assigned specific order of tasks and testing conditions. Participants are evenly and randomly distributed among these groups. | 12 |
| 2.3 | The distribution of participants’ opinions on AI-powered programming tools and their awareness of web accessibility. The percentages in the distribution column indicate the proportion of participants who either disagree (including both ‘strongly disagree’ and ‘disagree’) or agree (including both ‘strongly agree’ and ‘agree’) with the provided statements. | 13 |
| 2.4 | The (partial) chat history revealed that directly copying Copilot’s code suggestions would be incomplete, as developers overlooked the additional recommended steps. | 15 |
| 2.5 | Prompt instructions for the three LLM agents in CodeA11y | 20 |
| 2.6 | The distribution of opinions on AI-powered programming tools and their awareness of web accessibility based on the responses from participants in the evaluation study. The percentages in the distribution column indicate the proportion of participants who either disagree (including ‘strongly disagree’, ‘disagree’ and ‘slightly disagree’) or agree (including ‘strongly agree’, ‘agree’ and ‘slightly agree’) with the provided statements. | 23 |
| 2.7 | The distribution of participants’ opinions on GitHub Copilot and CodeA11y, as well as their ease of completing tasks with these tools. The distribution column shows the count of responses from Strongly Disagree (1) to Strongly Agree (7). | 25 |
| 3.1 | Metadata error analysis from a manual evaluation of 40 randomly selected papers from ACM CHI and ASSETS (2019-2024). | 42 |
| 3.2 | Comparison of PDF accessibility tagging between metadata generated by Acrobat’s auto-tagging feature, what authors submitted and our generated metadata. To compare the results, we manually evaluate 10 randomly selected papers from ACM ASSETS (2019-2024). | 42 |

Chapter 1

Introduction

More than one in four adults in the United States have some form of disability. Many disabilities affect computer use – for example visual disabilities can hinder the ability to see and interpret visual content, while physical disabilities can make it difficult to operate standard input devices like the mouse. To navigate these challenges, people with disabilities often use assistive technologies such as screen readers that read aloud a screen’s content for blind and low vision users. These technologies rely on essential metadata in the digital content, including alternative text for images, navigational structure, and other attributes that describe the interaction and purpose of interface components. Ensuring digital accessibility therefore requires authors to proactively provide this metadata that enables assistive technologies to interpret and interact with user interfaces effectively.

Unsurprisingly, authors fail to provide this metadata consistently, making most digital content inaccessible to people with disabilities. Less than $\sim 6\%$ home pages of the top million websites [40], 0.1% of more than a million tweets with images [87], $\sim 3\%$ of 20K scholarly PDFs [20], and 14% of the top COVID-19 data visualizations on the internet [45] are truly compliant with current accessibility standards [117]. Prior studies interviewing content authors across domains (*e.g.*, webmasters [126, 85], data practitioners [62] or researchers [64]) have invariably revealed a similar set of challenges: lacking awareness, time, incentive or tooling. Current accessibility tools that digital authors are expected to use offer inadequate automation for providing accessibility metadata. More often, authors must manually provide additional context to ensure proper interpretation of their content.

Recent advances in artificial intelligence (AI) offer an unprecedented opportunity to automate accessibility within the content authoring process. The rapid development of generative AI systems [46] enables this automation, and their widespread adoption [73] makes it possible to integrate these tools into existing workflows. Accordingly, this thesis claims:

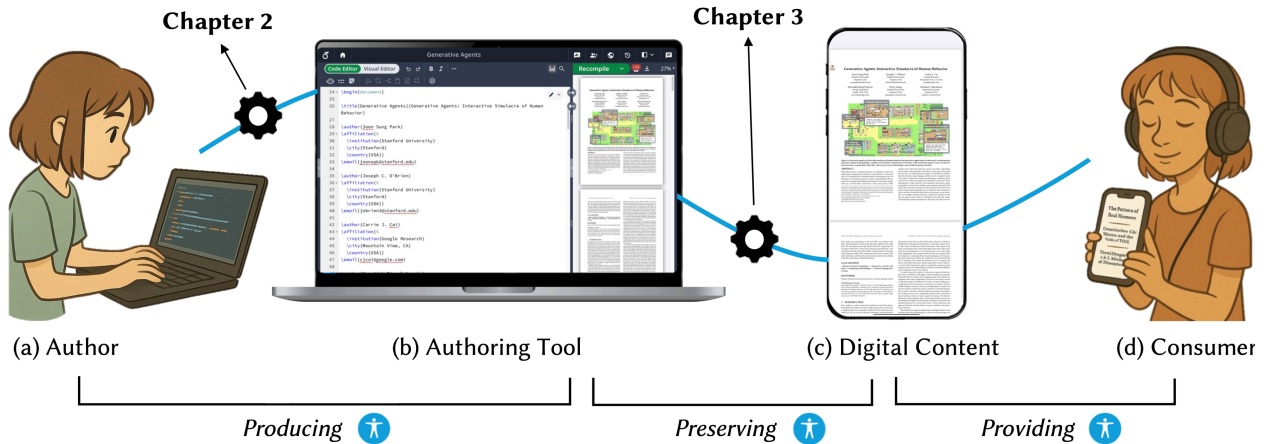


Figure 1.1: Accessibility metadata (👤) flows from (a) the author to (d) the consumer. The author produces metadata in (b) the authoring tool, which must preserve it in (c) the digital content. Assistive technologies then provide this metadata to the consumer. Chapters 2 and 3 detail tools that support the production and preservation of metadata respectively.

AI tools can enable developers and content creators to make their digital content accessible *automatically*.

1.1 Document Overview

Figure 1.1 illustrates the flow of accessibility metadata from the author to the consumer. For *providing* accessibility metadata, it must first be *produced* during authoring and then *preserved* during content rendering. In this thesis, I address two particularly stubborn challenges that disrupt this metadata flow in web and PDF accessibility:

- Web accessibility struggles with the *production* phase, where modern web frameworks generally provide features to support accessibility, but developers are ultimately responsible for implementing accessibility guidelines in their code.
- PDF accessibility is challenged in the *preservation* phase, where semantic structure frequently exists during authoring but is often discarded during PDF rendering.

To address these issues, I present two novel AI tools that assist authors in automatically producing and preserving accessibility metadata:

- **Chapter 2** introduces *CodeA11y*, an AI coding assistant that helps developers create user interface code that complies with accessibility standards.
- **Chapter 3** presents *WYSIWYM tagging*, a new PDF tagging method which preserves structure and reading order of source documents in the rendered PDFs.

On *Producing* Accessibility Metadata

Chapter 2

CodeA11y: AI Coding Assistant for Accessible Web Development

A persistent challenge in accessible computing is ensuring developers produce web UI code that supports assistive technologies. Despite numerous specialized accessibility tools, novice developers often remain unaware of them, leading to $\sim 95\%$ of web pages that contain accessibility violations [40]. AI coding assistants, such as GitHub Copilot, could offer potential by generating accessibility-compliant code, but their impact remains uncertain. Our formative study with 16 developers without accessibility training revealed three key issues in AI-assisted coding: failure to prompt AI for accessibility, omitting crucial manual steps like replacing placeholder attributes, and the inability to verify compliance. To address these issues, we developed CodeA11y, a GitHub Copilot Extension, that suggests accessibility-compliant code and displays manual validation reminders. We evaluated it through a controlled study with another 20 novice developers. Our findings demonstrate its effectiveness in guiding novice developers by reinforcing accessibility practices throughout interactions, representing a significant step towards integrating accessibility into AI coding assistants.

2.1 Preamble

This chapter is based on the following publications:

- [1] Peya Mowar. “Accessibility in AI-Assisted Web Development”. In: *Proceedings of the 21st International Web for All Conference*. 2024, pp. 123–125
- [2] Peya Mowar et al. “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. 2024, pp. 1–6

- [3] Peya Mowar et al. “CodeA11y: Making AI Coding Assistants Useful for Accessible Web Development”. In: *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 2025, pp. 1–15

2.2 Introduction

Most websites contain extensive accessibility errors [39], despite decades of investment in standards and guidelines [129, 116], tools [127, 108], advocacy [125, 63, 53], and policy. According to a recent analysis by WebAim [39], the homepages of the top million websites each contain 57 accessibility errors on average, including (but not limited to) missing alt-text for images [124, 83], inadequate color contrast [102], incorrect or missing labels for forms and links [110], and improper use of heading levels [115]. As a result, many people with disabilities will find it difficult to use these websites effectively and may not be able to use them at all.

Front-end web developers ultimately determine the accessibility (or inaccessibility) of the UI code that they write. Getting front-end developers to write more accessible code has proven exceptionally difficult. As Jonathan Lazar *et al.* [126] wrote twenty years ago in 2004, “Since tools and guidelines are available to help designers and webmasters make their web sites accessible, it is unclear why so many sites remain inaccessible.” A survey of webmasters at the time indicated that they generally would like to make their web pages accessible but cited a number of reasons they do not: “lack of time, lack of training, lack of managerial support, lack of client support, inadequate software tools, and confusing accessibility guidelines.” Sixteen years later, Patel *et al.* [85] reported remarkably similar results in their 2020 survey of 77 technology professionals. Few developers had received formal accessibility training, implementing accessibility was considered confusing, and advocating for accessible development was in conflict with other business goals. Clearly, what we have done so far is not working.

We argue that AI coding assistants (*e.g.*, Github Copilot [14]) could offer an opportunity to make UI code more accessible. They are already widely adopted, which means that developers do not need to be convinced to use them or to install a specialized tool for accessibility. They produce a wide variety of UI code and are capable enough to both reflect on the quality of arbitrary code and also prompt developers to fix what they are unable to do. This work explores how AI coding assistants currently help developers create UI code, what problems remain, and presents a system called CodeA11y that shows that AI coding assistants can be

made better at enabling developers to improve the accessibility of their UI code.

To explore this potential opportunity, we first conducted a user study (Section 2.4) with 16 developers not trained in accessibility to understand how current tools (GitHub Copilot) impact the production of accessible UI code. Our findings (Section 2.5) show that while Copilot may potentially improve accessibility of UI code, three barriers prevent realization of those improvements: (1) developers may need to explicitly prompt the assistants for accessible code and thus not benefit if they fail to do so, (2) developers may overlook critical manual steps suggested by Copilot, such as replacing placeholders in alternative text for images, and (3) developers may not be able to verify if they fully implemented more complex accessibility enhancements properly. The formative study showed the potential of AI coding assistants to improve the accessibility of UI code, but revealed several gaps that led us to design goals (Section 2.6) for improving AI coding assistants to support accessibility.

We then built *CodeA11y* (Section 2.7, Figure 2.3), a GitHub Copilot Extension that addresses the observed gaps by consistently reinforcing accessible development practices throughout the conversational interaction. We evaluated *CodeA11y* (Section 2.8) with 20 developers, assessing its effectiveness in supporting accessible UI development and gathering insights for further refinement. We found that developers using *CodeA11y* are significantly more likely to produce accessible UI code. Finally, we reflect on the broader implications of integrating AI coding assistants into accessibility workflows, including the balance between automation and developer education, and the potential for AI tools to shape long-term developer behavior toward accessibility-conscious practices (Section 2.9).

2.3 Related Work

Our research builds upon (i) Assessing Web Accessibility, (ii) End-User Accessibility Repair, and (iii) Developer Tools for Accessibility.

2.3.1 Assessing Web Accessibility

From the earliest attempts to set standards and guidelines, web accessibility has been shaped by a complex interplay of technical challenges, legal imperatives, and educational campaigns. Over the past 25 years, stakeholders have sought to improve digital inclusion by establishing foundational standards [129, 116], enforcing legal obligations [128, 107], and promoting a broader culture of accessibility awareness among developers [125, 63, 53]. Despite these longstanding efforts, systemic accessibility issues persist. According to the 2024 WebAIM Million report [39], 95.9% of the top one million home pages contained detectable WCAG

violations, averaging nearly 57 errors per page. These errors take many forms: low color contrast makes the interface difficult for individuals with color deficiency or low vision to read text; missing alternative text leaves users relying on screen readers without crucial visual context; and unlabeled form inputs or empty links and buttons hinder people who navigate with assistive technologies from completing basic tasks. Together, these accessibility issues not only limit user access to critical online resources such as healthcare, education, and employment but also result in significant legal risks and lost opportunities for businesses to engage diverse audiences. Addressing these pervasive issues requires systematic methods to identify, measure, and prioritize accessibility barriers, which is the first step toward achieving meaningful improvements.

Prior research has introduced methods blending automation and human evaluation to assess web accessibility. Hybrid approaches like SAMBA combine automated tools with expert reviews to measure the severity and impact of barriers, enhancing evaluation reliability [121]. Quantitative metrics, such as Failure Rate and Unified Web Evaluation Methodology, support large-scale monitoring and comparative analysis, enabling cost-effective insights [123, 25]. However, automated tools alone often detect less than half of WCAG violations and generate false positives, emphasizing the need for human interpretation [118, 106]. Recent progress with large pretrained models like Large Language Models (LLMs) [12, 42] and Large Multimodal Models (LMMs) [21, 43] offers a promising step forward, automating complex checks like non-text content evaluation and link purposes, achieving higher detection rates than traditional tools [23, 9]. Yet, these large models face challenges, including dependence on training data, limited contextual judgment, and the inability to simulate real user experiences. These limitations underscore the necessity of combining models with human oversight for reliable, user-centered evaluations [121, 106, 9].

Our work builds on these prior efforts and recent advancements by leveraging the capabilities of large pretrained models while addressing their limitations through a developer-centric approach. CodeA11y integrates LLM-powered accessibility assessments, tailored accessibility-aware system prompts, and a dedicated accessibility checker directly into GitHub Copilot—one of the most widely used coding assistants. Unlike standalone evaluation tools, CodeA11y actively supports developers throughout the coding process by reinforcing accessibility best practices, prompting critical manual validations, and embedding accessibility considerations into existing workflows.

2.3.2 End-user Accessibility Repair

In addition to detecting accessibility errors and measuring web accessibility, significant research has focused on fixing these problems. Since end-users are often the first to notice accessibility problems and have a strong incentive to address them, systems have been developed to help them report or fix these problems.

Collaborative accessibility [114, 111], enabled these end-user contributions to be scaled through crowd-sourcing. AccessMonkey [120] and Accessibility Commons [119] were two examples of repositories that store accessibility-related scripts and metadata, respectively. Other work has developed browser extensions that leverage crowd-sourced databases to automatically correct reading order, alt-text, color contrast, and interaction issues [113, 104]. One drawback of these approaches is that they cannot fix problems for an “unseen” web page on-demand, so many projects aim to automatically detect and improve interfaces without the need for an external source of fixes. A large body of research has focused on making specific web media (e.g., images [88, 94, 83, 82, 70], design [92, 90, 65, 54], and videos [86, 75, 74, 47]) accessible through a combination of machine learning (ML) and user-provided fixes. Other work has focused on applying more general fixes across all websites.

Opportunistic accessibility addressed a common accessibility problem of most websites: by default, content is often hard to see for people with visual impairments, and many users, especially older adults, do not know how to adjust or enable content zooming [105]. To this end, a browser script (`oppaccess.js`) was developed that automatically adjusted the browser’s content zoom to maximally enlarge content without introducing adverse side-effects (e.g., content overlap). While `oppaccess.js` primarily targeted zoom-related accessibility, recent work aimed to enable larger types of changes, by using LLMs to modify the source code of web pages based on user questions or directives [50].

Several efforts have been focused on improving access to desktop and mobile applications, which present additional challenges due to the unavailability of app source code (e.g., HTML). Prefab is an approach that allows graphical UIs to be modified at runtime by detecting existing UI widgets, then replacing them [109]. Interaction Proxies used these runtime modification strategies to “repair” Android apps by replacing inaccessible widgets with improved alternatives [99, 96]. The widget detection strategies used by these systems previously relied on a combination of heuristics and system metadata (e.g., the view hierarchy), which are incomplete or missing in the accessible apps. To this end, ML has been employed to better localize [80] and repair UI elements [81, 79, 59, 4].

In general, end-user solutions to repairing application accessibility are limited due to the lack of underlying code and knowledge of the semantics of the intended content.

2.3.3 Developer Tools for Accessibility

Ultimately, the best solution for ensuring an accessible experience lies with front-end developers. Many efforts have focused on building adequate tooling and support to help developers with ensuring that their UI code complies with accessibility standards.

Numerous automated accessibility testing tools have been created to help developers identify accessibility issues in their code: i) static analysis tools, such as IBM Equal Access Accessibility Checker [18] or Microsoft Accessibility Insights [27], scan the UI code’s compliance with predefined rules derived from accessibility guidelines; and ii) dynamic or runtime accessibility scanners, such as Chrome Devtools [15] or axe-Core Accessibility Engine [10], perform real-time testing on user interfaces to detect interaction issues not identifiable from the code structure. While these tools greatly reduce the manual effort required for accessibility testing, they are often criticized for their limited coverage. Thus, experts often recommend manually testing with assistive technologies to uncover more complex interaction issues. Prior studies have created accessibility crawlers that either assist in developer testing [35, 36] or simulate how assistive technologies interact with UIs [76, 57, 56].

Similar to end-user accessibility repair, research has focused on generating fixes to remediate accessibility issues in the UI source code. Initial attempts developed heuristic-based algorithms for fixing specific issues, for instance, by replacing text or background color attributes [60]. More recent work has suggested that the code-understanding capabilities of LLMs allow them to suggest more targeted fixes. For example, a study demonstrated that prompting ChatGPT to fix identified WCAG compliance issues in source code could automatically resolve a significant number of them [51]. Researchers have sought to leverage this capability by employing a multi-agent LLM architecture to automatically identify and localize issues in source code and suggest potential code fixes [26].

While the approaches mentioned above focus on assessing UI accessibility of already-authored code (*i.e.*, fixing existing code), there is potential for more proactive approaches. For example, LLMs are often used by developers to generate UI source code from natural language descriptions or tab completions [69, 14, 24, 16, 55, 61], but LLMs frequently produce inaccessible code by default [6, 30], leading to inaccessible output when used by developers without sufficient awareness of accessibility knowledge. The primary focus of this work is to design a more accessibility-aware coding assistant that both produces more accessible code without manual intervention (*e.g.*, specific user prompting) and gradually enables developers to implement and improve accessibility of automatically-generated code through IDE UI modifications (*e.g.*, reminder notifications).

2.4 Formative Study Methods

We conducted a formative study to assess the implications of AI coding assistants on web accessibility. We recruited novice developers and tasked them with editing real-world websites using GitHub Copilot. Our goal was to better understand how the use of Copilot affected the accessibility of the user interface code they produced.

2.4.1 Tasks

Table 2.1: Our formative study included four tasks. Each task was not primarily about accessibility but included an accessibility issue that was required to complete the task successfully. We adopt the scales of Unacceptable, Average and Good from prior work [68]. Uninformative attributes are those that merely reflect the field, such as ‘alt’ as alt-text or ‘click here’ as link description, without providing more meaningful or descriptive content [95]. Tasks are ranked from easy to difficult based on the time taken and success rates observed in our pilot studies.

| Task | Difficulty | Accessibility Issue | Evaluation Criteria |
|-----------------------|------------|---------------------|--|
| Button Visibility | Easy | Contrast | <i>Unacceptable:</i> contrast ratio of $< 4.5:1$ for normal text and $< 3:1$ for large text <i>Average:</i> WCAG level AA in default state (contrast ratio of $\geq 4.5:1$ for normal text) <i>Good:</i> WCAG level AA in all states (default, hover, active, focus, etc.) |
| Form Element | Moderate | Form Labeling | <i>Unacceptable:</i> Missing form labelling <i>Average:</i> One of form labels and keyboard navigation <i>Good:</i> Both form labeling and keyboard navigation |
| Add Section | Moderate | Link Labeling | <i>Unacceptable:</i> Missing link descriptions <i>Average:</i> Uninformative link descriptions <i>Good:</i> Descriptive link descriptions |
| Enhance Image for SEO | Difficult | Adding alt-text | <i>Unacceptable:</i> Missing or uninformative alt-text <i>Average:</i> Added alt-text with < 3 required descriptors [71] <i>Good:</i> Added alt-text with ≥ 3 out of 4 required descriptors |

The participants completed tasks in the codebases for two open-source websites, Kubernetes [33] and BBC News [32]. Both websites received over 2 million monthly visits worldwide [34] and belong to different categories in the IAB Content Taxonomy [17]. These websites were developed using different web development frameworks (Hugo and React,

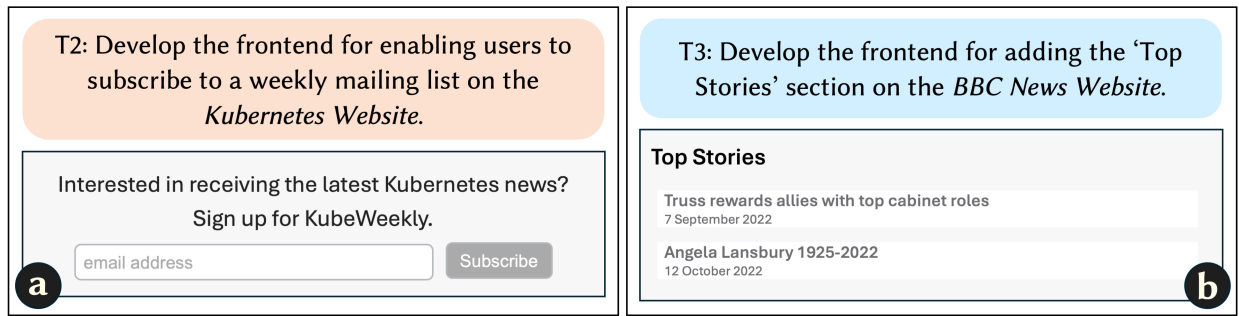


Figure 2.1: Examples of task descriptions and visual references given to our participants: (a) Task 2 was to implement a new contact form for subscribing to a mailing list, and (b) Task 3 was to add a ‘Top Stories’ section with linked articles. Successfully completing them required proper labeling of the form elements and links, but this was not explicitly stated in the instructions.

respectively). To choose the four specific tasks used in this formative study (Table 2.1), we sampled actual issues from each website’s repository. We chose issues for which accessibility needed to be considered to complete them correctly, but accessibility was not explicitly mentioned as a requirement in either the task description given to participants or on the issue description on the website’s code repository, as illustrated in Figure 2.1. Correctly performing the tasks required the consideration of several common web accessibility issues: color contrast, alternative text, link labels, and form labeling [39]. The goal was to mirror the kinds of specifications that developers often receive that do not explicitly mention accessibility.

2.4.2 Protocol

Our within-subjects user study had two conditions: (1) a control condition where participants received no AI assistance, and (2) a test condition where participants used GitHub Copilot. Each participant was assigned to edit two distinct websites, each with two tasks. To counterbalance order effects, participants were evenly and randomly assigned to one of four user groups (Table 2.2), balanced by website order and control/test conditions. Further, to simulate real-world scenarios, we concealed the true purpose of the study from participants. Participants were informed that the study was about the usability of AI pair programmers in web development tasks but were not explicitly instructed to make their web components accessible. This allowed us to observe how developers naturally handle accessibility when it is not explicitly emphasized, reflecting typical developer behavior. The research protocol was reviewed and approved by the Institutional Review Board (IRB) at our university.

Table 2.2: Participant User Groups: Each group is assigned specific order of tasks and testing conditions. Participants are evenly and randomly distributed among these groups.

| # | Order 1, Testing Condition | Order 2, Testing Condition |
|---|--------------------------------|--------------------------------|
| 1 | Kubernetes, With AI Assistance | BBC News, No AI Assistance |
| 2 | Kubernetes, No AI Assistance | BBC News, With AI Assistance |
| 3 | BBC News, With AI Assistance | Kubernetes, No AI Assistance |
| 4 | BBC News, No AI Assistance | Kubernetes, With AI Assistance |

2.4.3 Participants






We employed convenience sampling and snowball sampling methods to recruit our participants. Our study was advertised on university bulletin boards, social media, and shared communication channels (Twitter, Slack, and mailing groups). Our recruitment criteria stipulated that participants must be over 18 years of age, live in the United States, and have self-assessed familiarity with web development. Further, we required the participants to be physically present on our university campus for the duration of the study. To avoid priming during participant recruitment, we did not stipulate awareness of web accessibility as an eligibility criterion. We chose university-specific avenues for recruiting CS students, that reflect a typical novice developer cohort.

Our study enlisted 16 participants (7 female and 9 male; ages ranged from 22 to 29). Almost all of our participants were students and had multiple years of coding experience. Most (n=10) had multi-year *industrial* programming experience (e.g., full-time or intern experience in the company). Nearly all participants (except one) had previously used AI coding assistants. GitHub Copilot and OpenAI ChatGPT were the most popular (n=10). Others preferred Tabnine (n=6) and AWS CodeWhisperer (n=2). 12 participants had self-described substantial experience with HTML and CSS. 10 were proficient in JavaScript and 7 were proficient in React.js. Despite this expertise, the majority (14 participants) were unfamiliar with the Web Content Accessibility Guidelines (WCAG). Only 2 participants knew about these guidelines, but they had not actively engaged in creating accessible web user interfaces or received formal training on the subject (details are provided in Table 2.3).

2.4.4 Procedure

The study was conducted in person at our lab, where participants performed programming tasks on a MacBook Pro laptop equipped with IntelliJ IDEA with the GitHub Copilot plugin preinstalled. Before starting the study, we explained the study procedure to the

Table 2.3: The distribution of participants’ opinions on AI-powered programming tools and their awareness of web accessibility. The percentages in the distribution column indicate the proportion of participants who either disagree (including both ‘strongly disagree’ and ‘disagree’) or agree (including both ‘strongly agree’ and ‘agree’) with the provided statements.

| Statement | Distribution | | | | |
|--|--------------|---|-----|--|--|
| “I trust the accuracy of AI programming tools.” | 13% |  | 25% | | |
| “I am proficient in web accessibility.” | 75% |  | 19% | | |
| “I am familiar with the web accessibility standards, such as WCAG 2.0.” | 88% |  | 12% | | |
| “I am familiar with ARIA roles, states, and properties.” | 69% |  | 25% | | |
|  | | | | | |

participants and took their informed consent. The participants then watched a 5-minute instructional video explaining Copilot’s features, such as code autocompletion and the Copilot chat¹. Participants were assigned tasks related to two selected websites, with a total of four tasks to complete in 90 minutes. They were required to work on one website with and the other without GitHub Copilot. Further, they were allowed to access the web for task exploration or code documentation through traditional search engines like Google Search, but with generative results turned off. During the coding session, a researcher observed silently, offering help with tasks, tool usage, or debugging only if participants were stuck, and asked them to move on after 30 minutes, without giving any accessibility-related hints. Based on our observations from pilot studies, we set time limits ranging from 15 to 30 minutes per task. Finally, after completing the coding tasks, they were asked to participate in a 10-15 minute survey on their experience in AI-assisted programming and web accessibility, development expertise, and open-ended feedback. In the end, the participants were compensated with a gift voucher worth 30 USD.

2.4.5 Data Collection and Analysis

We collected both quantitative and qualitative data for a mixed-method analysis. For quantitative data, we used an IntelliJ IDEA plugin [11] that tracked user actions — such as keyboard input (typing, backspace), IDE commands (copy, paste, undo), and interactions with GitHub Copilot (accepting suggestions, opening the Copilot Chat window) — and recorded their timestamps. Additionally, we employed the axe-Core Accessibility Engine 2 to gather accessibility violation metrics, including the type and count of WCAG failures, for

¹<https://www.youtube.com/watch?v=jXp5D5ZnxGM>

each code submission, a method proven reliable in previous studies [52]. We also collected AI usage, programming languages and framework preferences, and expertise in web accessibility via a post-task survey.

On the qualitative side, we captured the entire study sessions through screen recordings, resulting in a total of 18.73 hours of video data. We complemented this with observational notes taken during the sessions, documenting verbal comments made by participants. The participants' interactions with Copilot Chat were also recorded for further analysis between prompts and the final code. The analysis of this data was carried out using open coding and thematic analysis [97]. Some themes that emerged were: 'visual enhancement', 'recalling syntax', 'feature request', and 'code understanding'. For accessibility evaluation, we manually inspected the websites created during the study and evaluated their accessibility on a qualitative scale of "Unacceptable," "Average," and "Good" adopted from prior work [68]. The criteria for these evaluations were developed per best practices identified in prior research published in CHI and ASSETS, detailed further in Table 2.1.

2.5 Formative Findings

Our formative study revealed that while existing AI coding assistants can produce accessible code, developers still need accessibility expertise for effective use. Otherwise, (1) the accessibility introduced is likely to not be applied comprehensively, (2) the advanced features recommended by the assistant are unlikely to be implemented, (3) the accessibility errors introduced by the assistant are unlikely to be caught.

2.5.1 Developer Behavior

In the study, participants spent slightly more time on tasks without Copilot, averaging 30.84 minutes ($\sigma = 11.95$) compared to 28.94 minutes ($\sigma = 8.57$) with Copilot. Copilot also facilitated a greater volume of code edits (13.28 lines of code, $\sigma = 9.02$ vs 10.41 lines of code, $\sigma = 5.87$), indicating that AI-assisted workflows encouraged iterative coding practices. However, even with Copilot, participants spent approximately 39.84% of their task time (11.91 minutes, $\sigma = 8.00$) away from the IDE, browsing the web or checking the rendered HTML, highlighting the importance of traditional validation methods. The study also found fewer backspace key presses, an indicator of post-paste corrections, without Copilot ($\mu = 92.62$, $\sigma = 68.27$) than with the AI assistant ($\mu = 104.50$, $\sigma = 91.91$). Further, code pasting was slightly higher when participants solely browsed the web for exploration, averaging 12.68 times ($\sigma = 8.09$), compared to 11.43 times ($\sigma = 5.35$) with access to Copilot Chat. Participants

dedicated about 7.39% (2.14 minutes, $\sigma = 1.72$) of their task time typing in the GitHub Copilot chat window, while, they also accepted Copilot’s code auto-complete suggestions around 5.44 times ($\sigma = 5.00$) on average.

2.5.2 AI Usage and Prompting Strategies

Participants mainly used the autocomplete feature only when they had a clear mental model of the desired code structure and sought to accelerate the code typing process. In contrast, they heavily relied on the conversational interface for syntax inquiries, conceptual understanding, and the generation of code templates. We noticed that our participants wrote brief, task-oriented prompts that focused on immediate code solutions or specific interface modifications, often disregarding broader architectural considerations. Their prompting style was iterative and reactive, frequently requesting small incremental changes, fixes to previous outputs, or refinements to their vague prompts.

Table 2.4: The (partial) chat history revealed that directly copying Copilot’s code suggestions would be incomplete, as developers overlooked the additional recommended steps.

| | |
|-----------------|--|
| User: | “User Query” |
| Copilot: | Generated Code Associated Text Explanation |
| User 1: | “I want to label the images or add relevant image descriptors” |
| Copilot: | alt=imgAlt // Add this line Remember to replace imageAlt, <...> with your actual image attributes. In this example, imageAlt is the alt-text for the image, and <...>. You will need to replace these with the actual values. |
| User 2: | “Can you optimise the images on this article page for search engine optimization. Refactor the code accordingly by labeling the images or adding relevant image descriptions.” |
| Copilot: | imgAlt=imageDescription // Add this line Sure, to optimize images for SEO, you can add alt attributes to your img tags. The alt attribute provides alternative information for an image if a user for some reason cannot view it (because of slow connection, an error in the src attribute, or if the user uses a screen reader). |

Furthermore, none of the participants, including the two who were familiar with web accessibility, prompted with accessibility in mind. Instead, our participants’ prompts centered around visual and functional attributes (e.g., “add a gray background to the subscription form” (P4) or “add a grey patch” (P1)). Consequently, the AI assistant’s suggestions often failed to incorporate accessibility best practices automatically. Occasionally, our participants prompted

for enhancements that indirectly aligned with accessibility requirements, and Copilot provided relevant accessibility suggestions, as shown in Table 2.4. However, participants’ overreliance on AI assistance often led them to assume that Copilot’s code output was correct and complete. For instance, despite additional explanations from Copilot advising manual adjustments to image descriptions, participants directly pasted the code, resulting in code submissions with empty `alt` attributes.

2.5.3 Implications for Web Accessibility

Our study showed mixed results of AI assistants in considering accessibility issues with no statistically significant difference between the experimental conditions, as shown in Figure 2.2. Notably, Copilot could (sporadically) generate accessible components by utilizing patterns from other parts of a website. For example, it might automatically include proper labels for form fields, such as `<label for="email"> Email: </label>` in a signup form. However, there were also instances where Copilot inadvertently introduced new accessibility issues. For example, when adding new button components with hover effects, it failed to ensure adequate contrast between the hover color and background.

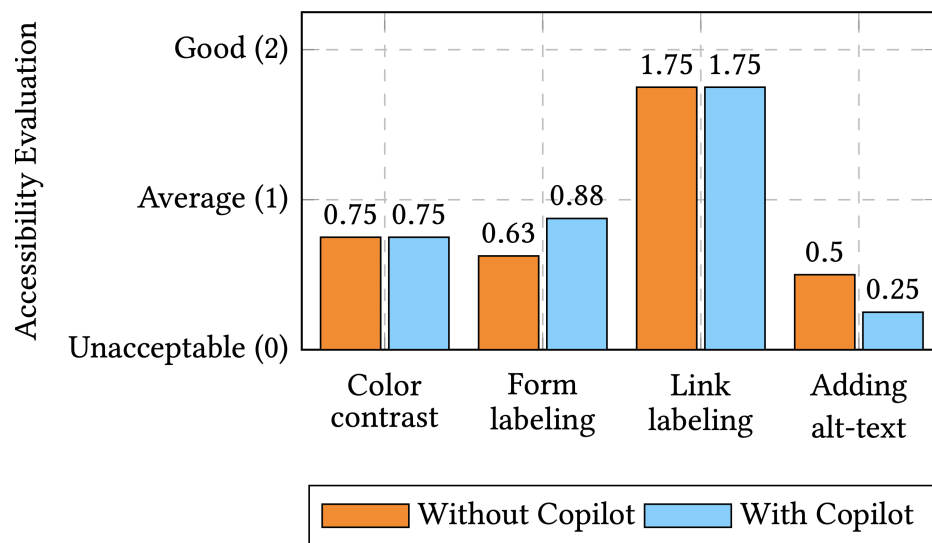


Figure 2.2: Mean Accessibility Evaluation Scores by Tasks and Copilot Usage: Higher scores indicate success.

Further, the effectiveness of AI assistants was limited by the need for more sophisticated accessibility knowledge. Since our participants had limited awareness about these accessibility features, they would often ignore such suggestions by manually deleting the `<label>` tag or blindly accepting `alt = "" // Add your text here`. Some errors, such as providing

blank alt-texts for informative images, were not even flagged by automated accessibility checkers because they interpret the image as decorative and consider this deliberate. This is particularly problematic as it implies that AI assistance might increase the risk of accessibility oversights, allowing critical errors to go unnoticed and uncorrected.

2.6 Design Requirements

Our formative study identified three limitations in novice developers' interactions with AI assistants: (1) failing to prompt for accessibility considerations explicitly, (2) uncritically accepting incomplete code suggestions from Copilot, and (3) struggling to detect potential accessibility issues in their code. These shortcomings indicate possible directions to support accessibility through three design goals (**G1-G3**):

2.6.1 (G1) Integrate System Prompts for Accessibility Awareness

Without explicit prompting, the AI assistant rarely produced accessibility-compliant code, reflecting the accessibility issues prevalent in its training data. However, it occasionally suggested accessibility features when participants indirectly prompted them, demonstrating its ability to recall accessibility practices from training data upon instruction. AI assistants should have a system prompt tuned towards following accessibility guidelines by default, for consistent generation of accessibility-compliant code, even when developers do not mention accessibility specifically. Further, the system prompt should also direct the assistant to suggest accessibility-focused iterative refinements.

2.6.2 (G2) Support Identification of Accessibility Issues

Due to their unfamiliarity with accessibility standards, our participants were unable to identify compliance issues in the existing and modified code. They primarily prompted changes to individual components (such as buttons and forms), hardly addressing broader page-level accessibility concerns (such as heading structure or landmark regions). AI assistants should not only automatically generate accessibility-compliant code, but also provide real-time feedback to detect and resolve accessibility violations within the codebase. Further, AI assistants and accessibility checkers should work in tandem to ensure that incomplete or incorrect implementations of the AI-suggested code are always flagged by the latter.

2.6.3 (G3) Encourage Developers to Complete AI-Generated Code

Our observations revealed that accessibility implementation in AI-assisted coding workflows commonly required critical manual intervention to complete and validate AI-generated code. This involved replacing placeholder attributes, such as labels and alt-texts, with meaningful values and verifying color contrast ratios. However, we found that participants blindly copy-pasted code and proceeded further if there were no apparent errors. This behavior of deferring thought to suggestions has also been documented in previous work [31]. To mitigate this, AI assistants should proactively remind developers to ensure that all necessary accessibility features – such as contrast ratios or keyboard navigation support – are fully implemented and verified.

2.7 CodeA11y System Overview

Guided by the design goals identified through our user study, we built CodeA11y, a GitHub Copilot Extension for Visual Studio IDE (see Figure 2.3). In this section, we present the interactions that it supports and its system architecture.

CodeA11y has three primary features (**F1-F3**, aligned to G1-G3, respectively): (**F1**) it produces user interface code that better complies with accessibility standards, (**F2**) it prompts the developer to resolve existing accessibility errors in their website, and (**F3**) it reminds the developer to complete any AI-generated code that requires manual intervention. CodeA11y is integrated into Visual Studio Code as a GitHub Copilot Extension², enabling CodeA11y to act as a chat participant within the GitHub Copilot Chat window panes. While we implemented this as an extension, it could be integrated directly into Copilot in the future.

2.7.1 Multi-Agent Architecture

CodeA11y has three LLM agents (Figure 2.4): Responder Agent, Correction Agent, and Reminder Agent. We provide their prompt instruction highlights in Table 2.5. These agents facilitate each of the above features (F1-F3) as follows:

- **Responder Agent** (for F1): This agent generates relevant code suggestions based on the developer’s prompt. It assumes that the developer is unfamiliar with accessibility standards and automatically generates accessible code. The prompt instruction for this

²<https://docs.github.com/en/copilot/using-github-copilot/using-extensions-to-integrate-external-tools-with-copilot-chat>

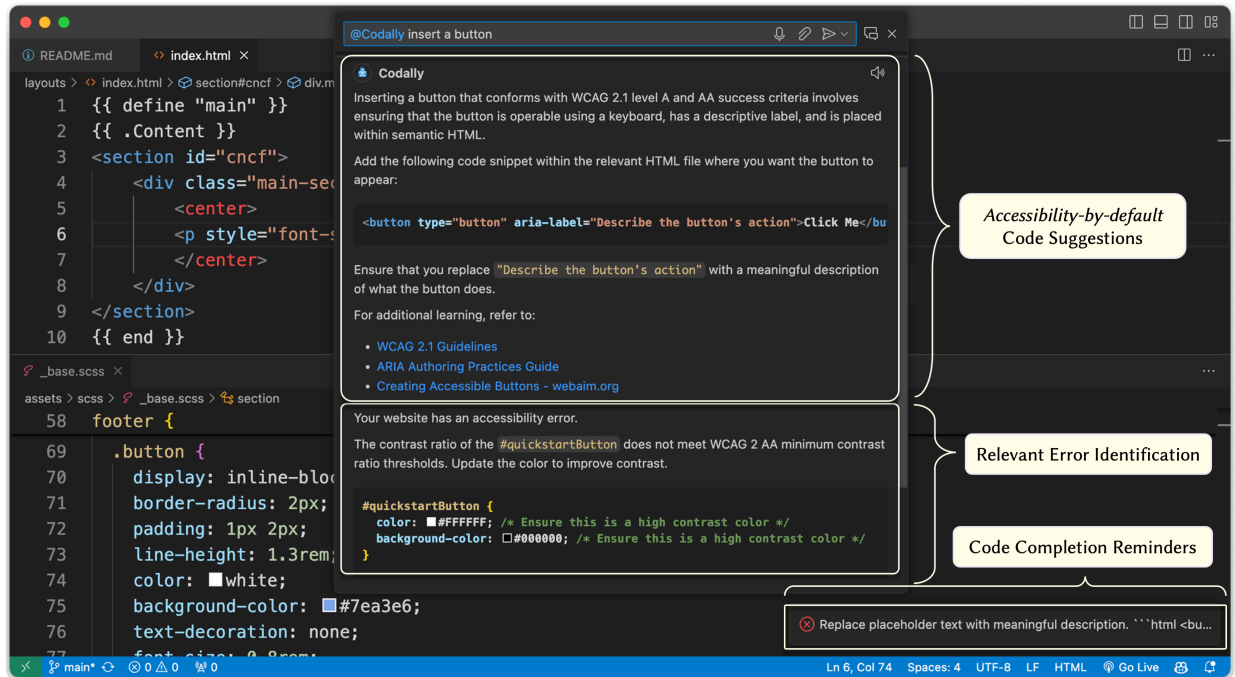


Figure 2.3: CodeA11y is a GitHub Copilot Extension for Accessible Web Development. CodeA11y addresses accessibility limitations of Copilot observed in our study with developers through three features: (1) accessibility-by-default code suggestions, (2) automatic identification of relevant accessibility errors, and (3) reminders to replace placeholders in generated code. Integrating these features directly into AI coding assistants would improve the accessibility of the user interfaces (UIs) developers create.

agent is adapted from GitHub’s recommended user prompt for accessibility.³

- **Correction Agent** (for F2): This agent parses through accessibility error logs produced by an automated accessibility checker (axe DevTools Accessibility Linter⁴) to hint the developer at making additional accessibility fixes in the component or page being currently discussed in the chat context.
- **Reminder Agent** (for F3): This agent reviews the Responder Agent’s suggestions and identifies required manual steps for completing their implementation. It accordingly sends reminder notifications to the developer through the Visual Studio IDE infrastructure.

The agents are provided with several different sources of context:

- **Code Context**: 100 lines of code centered on the cursor position in the active files.

³<https://github.blog/developer-skills/github/prompting-github-copilot-chat-to-become-your-personal-ai-assistant-for-accessibility/>

⁴<https://www.deque.com/axe/devtools/linter/>

Table 2.5: Prompt instructions for the three LLM agents in CodeA11y

| Agent | Prompt Instruction Highlights |
|------------------|--|
| Responder Agent | <ul style="list-style-type: none"> • I am unfamiliar with accessibility and need to write code that conforms with WCAG 2.1 level AA criteria. • Be an accessibility coach that makes me account for all accessibility requirements. • Use reputable sources such as w3.org, webaim.org and provide links and references for additional learning. • Don't give placeholder variables but tell me where to give meaningful values. • Prioritise my current request and don't mention accessibility if I give a generic request like "Hi". |
| Correction Agent | <ul style="list-style-type: none"> • Review the accessibility checker log and provide feedback to fix errors relevant to current chat context. • If a log error relevant to current chat context occurs, provide a code snippet to fix it. |
| Reminder Agent | <ul style="list-style-type: none"> • Is there an additional step required by the developer to meet accessibility standards after pasting code? • Reminder should be single line. Be conservative in your response, if not needed, say "No reminders needed." • For example, remind the developer to replace the placeholder attributes with meaningful values or labels, or visually inspect element for colour contrast when needed. |

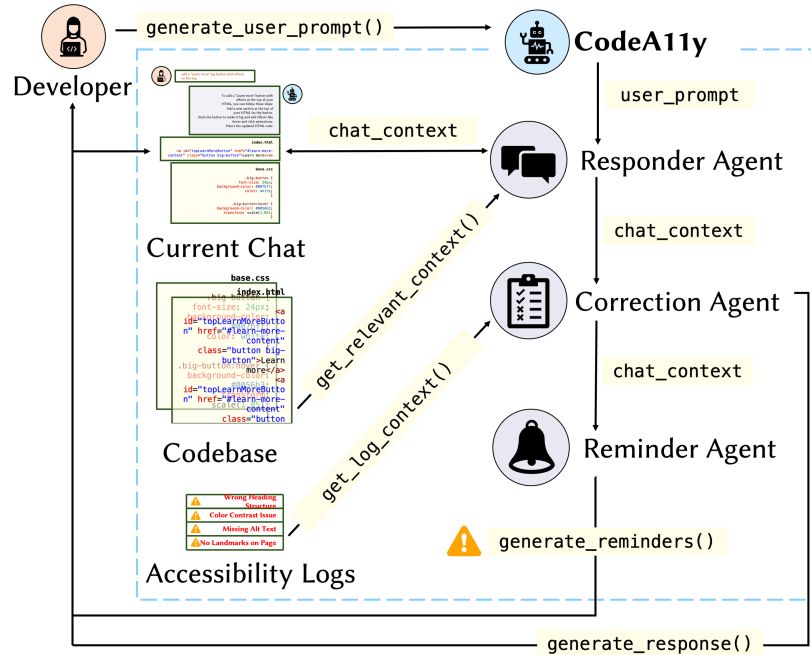


Figure 2.4: CodeA11y Architecture: Multi-agent workflow

- **Chat Context:** the current active chat window interaction.
- **Accessibility Linter Logs:** automated Axe DevTools Accessibility Linter error logs, refreshed periodically.
- **Project Context:** code context from the `README` and `index` files, which contain information about the web framework that is being used, information about project structure, and other key configuration details.

Due to the constraints in the context window, we optimized our prompts and filtered this context when it exceeded 4000 characters. The agents use GPT-4o as the back-end model from the same OpenAI GPT family of models that powers GitHub Copilot.

2.7.2 User Interaction

Developers invoke⁵ CodeA11y in the GitHub Copilot Chat window panes (includes Quick Chat and Chat View) using `@CodeA11y`. When a developer prompts CodeA11y, an internal `chat_context` state is established, storing the latest user prompts and agent responses. The `get_relevant_context()` function is called, which passes the source code and project context to the Responder Agent. The agent generates code suggestions, accessibility explanations, and links to additional resources and updates `chat_context`. The `get_log_context()`

⁵The goal is for GitHub Copilot to invoke CodeA11y automatically during frontend development tasks.

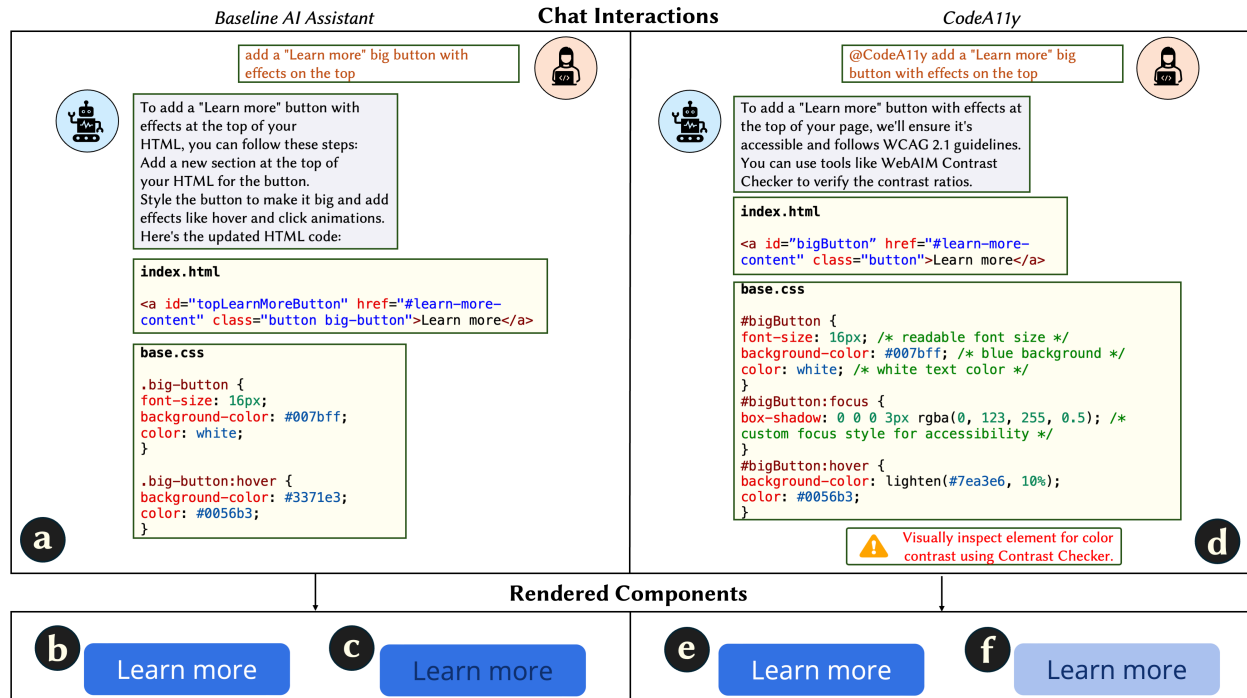


Figure 2.5: Contrasting responses for the same task across AI-assistants, showing differences in workflows. Developers had access to both the code and the rendered user interface. (a) and (d) represent conversations with the baseline assistant, and CodeA11y respectively. (b) and (e) show the buttons generated by each assistant in their default state. (c) and (f) display the buttons when hovered over, illustrating the differences in button color contrast.

function is called, which passes the accessibility linter logs to the Correction Agent. This agent suggests additional fixes and displays the responses in the chat pane. Lastly, the updated `chat_context` state is forwarded to the Reminder Agent, which generates and sends reminder notifications. Figure 2.5 illustrates a typical interaction between a developer and CodeA11y, showing how it compares to baseline assistants like GitHub Copilot.






2.8 User Evaluation

We conducted a within-subjects user study with 20 new participants to evaluate CodeA11y's effectiveness in guiding novice developers toward adhering to accessibility standards.

2.8.1 Methodology

We made the following revisions to our formative study protocol (Section 2.4). First, the experimental conditions were updated as follows: (1) the control condition involved using the baseline AI assistant (GitHub Copilot), and (2) the test condition where the participants used

Table 2.6: The distribution of opinions on AI-powered programming tools and their awareness of web accessibility based on the responses from participants in the evaluation study. The percentages in the distribution column indicate the proportion of participants who either disagree (including ‘strongly disagree’, ‘disagree’ and ‘slightly disagree’) or agree (including ‘strongly agree’, ‘agree’ and ‘slightly agree’) with the provided statements.

| Statement | Distribution | | |
|--|--------------|---|-----|
| “I trust the accuracy of AI programming tools.” | 15% |  | 70% |
| “I am proficient in web accessibility.” | 70% |  | 25% |
| “I am familiar with the web accessibility standards, such as WCAG 2.0.” | 80% |  | 15% |
| “I am familiar with ARIA roles, states, and properties.” | 85% |  | 10% |
|  | | | |

CodeA11y. Second, we changed the post-task survey to a brief semi-structured interview to get more nuanced insights about the usability of our system. We analyzed interview responses to better understand the factors shaping participants’ assistant preferences and their perceptions of any new coding practices introduced during the study. Third, we used Visual Studio Code as the IDE interface (which had advanced AI updates since the formative study – regarding both model performance and introduction of new features such as Inline Chat). Finally, we recruited 20 new participants for this subsequent study, with no prior exposure to the formative study, to evaluate CodeA11y’s performance on the same UI development tasks. These participants were the same demographic as our formative participants (students; multiyear programming experience; 6 female and 14 male; ages ranged from 22 to 30). Again, most participants were unfamiliar with the web accessibility standards (Table 2.6), but most (90%) had experience using AI programming tools. The IRB approved our protocol.

To avoid biasing participants towards adhering to accessibility guidelines, we did not disclose the specific purpose of the CodeA11y plugin. For the duration of the study, we renamed the assistant “Codally” and described it as a general-purpose chat assistant for website editing. We assumed the interface would be intuitive, similar to widely used assistants, and therefore briefed participants only on basic AI assistant usage (e.g., Copilot), deliberately withholding explanations of error pop-ups to prevent influencing their behavior before the main study tasks. However, during the course of our study, we realized that VS Code was dismissing popup boxes created by our plugin more rapidly than expected – causing some participants to miss them. After 8 participants, we switched from floating popups to modals (which prevent the IDE’s auto-dismissal) due to a technical limitation. Both notification

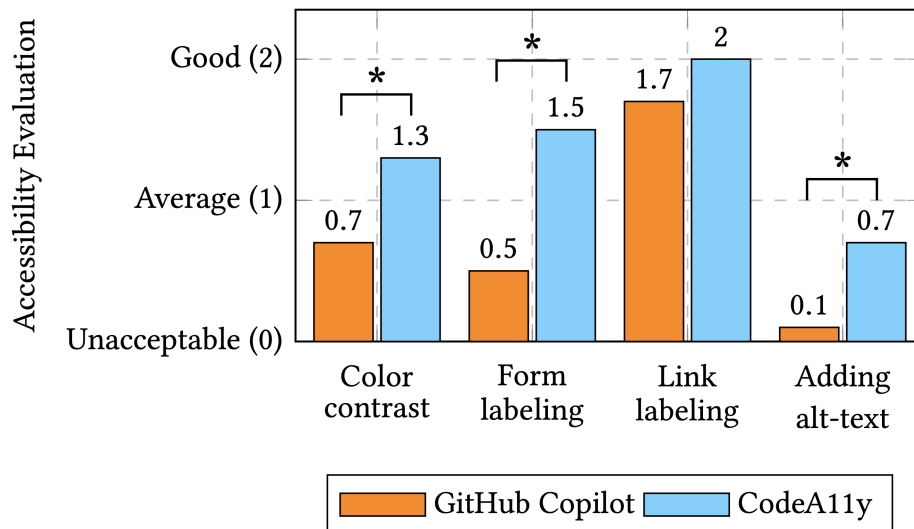


Figure 2.6: Mean Accessibility Evaluation Scores by Tasks and AI Assistant: Higher scores indicate success.

strategies do not require users to address errors, making them valid design choices. In our baseline comparison, we aggregate data from all users and include anecdotal observations of user behavior with each strategy. We acknowledge that such UI design choices may introduce variability and plan to investigate this further in future work.

2.8.2 Results

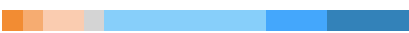
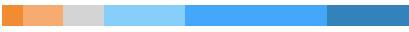
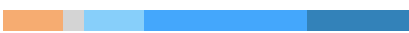
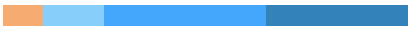
Here, we present the results of our subsequent evaluation study.

Accessibility Improvements

We implemented the accessibility assessments using the same measures outlined in our formative study (Table 2.1). Notably, our participants demonstrated a marked improvement in generating accessible web components and resolving accessibility issues with CodeA11y (Figure 2.6). CodeA11y facilitated the automatic addition of form labels and ensured contrasting colors for button states, leading to statistically significant enhancements in accessibility outcomes. Specifically, participants performed better at adding form labels ($\mu = 1.5$, $\sigma = 0.85$) compared to GitHub Copilot ($\mu = 0.5$, $\sigma = 0.85$; $t = 2.63$, $p < 0.05$) and in ensuring contrasting button colors ($\mu = 1.3$, $\sigma = 0.67$ vs. $\mu = 0.7$, $\sigma = 0.82$; $t = 1.78$, $p < 0.05$). We also observed improvements in adding alt-texts with CodeA11y ($\mu = 0.7$, $\sigma = 0.95$ vs. $\mu = 0.1$, $\sigma = 0.32$; $t = 1.9$, $p < 0.05$). Though we did not find any statistical improvements in labeling links (perhaps because GitHub Copilot did a decent job at this task

itself), all participants who used CodeA11y successfully completed this task ($\mu = 2$, $\sigma = 0$ vs. $\mu = 1.7$, $\sigma = 0.67$; $t = 1.9$, $p = 0.09$).

Table 2.7: The distribution of participants’ opinions on GitHub Copilot and CodeA11y, as well as their ease of completing tasks with these tools. The distribution column shows the count of responses from Strongly Disagree (1) to Strongly Agree (7).

| Statement | Distribution | | | | | |
|--|--------------|--|--|--|--|-----|
| “I am satisfied with the code suggestions provided by”: | | | | | | |
| GitHub Copilot | 20% |  | | | | 75% |
| CodeA11y | 15% |  | | | | 75% |
| “I found it easy to complete the coding tasks with”: | | | | | | |
| GitHub Copilot | 15% |  | | | | 80% |
| CodeA11y | 10% |  | | | | 90% |
| <div><div>Strongly Disagree</div><div>Disagree</div><div>Slightly Disagree</div><div>Neutral</div><div>Slightly Agree</div><div>Agree</div><div>Strongly Agree</div></div> | | | | | | |

Developers’ Perspectives

Overall, participants reported no statistically significant difference in satisfaction ($\mu = 5.15$, $\sigma = 1.75$ vs. $\mu = 4.95$, $\sigma = 1.67$; $t = 0.37$, $p = 0.36$) and ease of use ($\mu = 5.8$, $\sigma = 1.47$ vs. $\mu = 5.4$, $\sigma = 1.67$; $t = 0.8$, $p = 0.21$) between CodeA11y and Github Copilot (Table 2.7).

During the post-study interviews, participants provided additional reasoning for their preferences. Most (n=16) participants did not have a specific preference between the two assistants, which is consistent with the conclusion of our statistical analysis. Others did indicate a preference (n=3) but provided reasoning that was based on the complexity of the task rather than assistant features, “*I liked the first assistant (CodeA11y) better, maybe because of the tasks. The second one (GitHub Copilot) required me to understand the code, and the first directly gave me the code. That’s the difference.*” (P18)

We asked our participants if they were introduced to any new coding practices by either of the assistants. To our surprise, only 4 participants mentioned accessibility, demonstrating CodeA11y’s effectiveness in “silently” improving the accessibility of our participants’ UI code. These participants noted that they had not these considerations before. However, some mentioned either not paying attention to them or subconsciously rejecting them, as they were primarily focused on completing the tasks, which they considered unrelated to accessibility:

“I did not find any difference (between the assistants). When I was prompting CodeA11y, it was hinting at me to use alt texts, which was not happening in

Copilot. It didn't come to me by default, so that was good ... But I don't think I implemented that.” (P27)

Still, they appreciated CodeA11y for emphasizing best practices for accessibility. For instance, **P27** continued: *“I did see a few of the popups, and they did mention some interesting points like you need to consider the color of the button when you add a new button because if people are color blind, they might not be able to notice it.”* Further, **P19**, familiar with web accessibility but not proficient, realized that although he did not learn anything new about CodeA11y's color contrast suggestion, he noticed a visible difference in user experience after accepting it. Our sole participant who claimed proficiency in accessibility valued support for a specific framework:

“I am familiar with accessibility coding practices but not in the React Native environment; I don't know if I would have needed that help in HTML, but I liked that it tried to highlight accessibility practices in React Native.” (P21)

Other Observations

Participants frequently made minor edits to the AI-generated code for refining the visual appearance of web components. For instance, **P18** remarked, *“CodeA11y did the job for me; I only had to change the property values.”* However, while focusing on visual adjustments, participants occasionally removed accessibility enhancements suggested by CodeA11y. Further, many participants still overlooked the manual validation steps required for implementing more advanced accessibility features. Participants appeared to consistently lack interest in the floating popups, so 50% of participants using CodeA11y still added uninformative alt-texts. We observed a slightly different pattern with the modal reminders. Our participants initially paid attention to the modal interface, but then began to just close them. Although performance across the two reminder types is hard to assess objectively due to the small sample size, we observed higher means for the alt-text and form label tasks for the modal reminders. As a whole, the reminders proved somewhat effective: none of the participants using CodeA11y submitted empty alt-texts, which meant at least automated accessibility checkers would not consider the images decorative.

2.9 Discussion

This work has explored how AI coding assistants currently contribute to UI code that is accessible to people with disabilities. While these tools offer a new opportunity for achieving accessibility, we have revealed the remaining challenges and showed how they could be

addressed with changes to the way the coding assistants operate.

2.9.1 Which comes first: Adoption or Awareness?

Adoption is a perennial challenge faced by most accessibility technologies, even when the technology could lead to substantial improvements in user experience. One reason adoption is low is because awareness is low – people who could benefit from access technology do not know about it. For example, a survey found that only 10% of older adults knew what the term “accessibility” meant and therefore did not enable any useful settings [78, 91]. Similarly, developers benefit in many ways from tools that improve the accessibility of their code (*e.g.*, linters, scanners), resulting in better-designed applications and reaching more users. Unfortunately, many developers are unaware of these tools or unwilling to adopt new practices that require changing their original workflow. For example, while AI assistants like Copilot are capable of generating accessible code, our formative study found that developers were unaware or unwilling to explicitly prompt it to do so.

One goal of our work was to investigate whether developers could increase the adoption of accessibility technology and development practices independently or in tandem with awareness. For example, prior work [105] found that by “opportunistically” zooming into web pages and configuring settings, users could automatically benefit from improved accessibility. Our motivation is similar: we aimed to improve code accessibility while introducing minimal changes to existing AI-assistant developer workflows *i.e.*, GitHub Copilot. According to the Visual Studio Marketplace, GitHub Copilot has been installed over 20 million times (as of the time of writing), suggesting that many developers are already familiar with the plugin’s interactions, tooling, and interface. Our results show that CodeA11y significantly improved code accessibility while maintaining a similar (slightly improved) ease of use to Copilot. This suggests that if GitHub Copilot included our set of features or were willing to use a similar plugin without requiring substantial deviation from existing workflows, millions of developers could start writing more accessible code immediately.

2.9.2 AI-Assisted but Developer-Completed

Even when developers adopt accessibility tools, additional expertise is required to maximize their utility. This is especially true for AI assistants, which are incapable of generating entirely correct or accessible code. Our work offers some insight into the manual effort needed to write accessible code. Both our formative and evaluation studies underscore the necessity for developers to manually intervene with AI-generated code to effectively implement accessibility features. A recurrent challenge in AI-assisted coding is the generation of incomplete or

boilerplate code, which often requires developers to take additional steps for completion and validation. Our findings reveal that novice developers tend to critically evaluate AI outputs in areas they prioritize, such as visual enhancements, while overlooking aspects they are less familiar with, like accessibility.

One of the tensions of this work is that while we aim to increase the adoption of AI-driven accessibility tools among users with little expertise or awareness, some degree of understanding is required to use these tools effectively. CodeA11y and other tools can employ strategies to scaffold this interaction, *e.g.*, asking a user “can you describe what’s in this image?” instead of asking them directly for “alternative text.” However, developers ultimately need to be willing to expend additional effort and manually implement the more challenging aspects of this work. Thus, while our work suggests that it is possible to “silently” improve the accessibility of developer-written code, it is ultimately not a replacement for better accessibility awareness, development practices, and education. Nevertheless, CodeA11y could help gradually improve awareness by slowly introducing and explaining accessibility concepts to users after they have found benefits from using the tool.

2.9.3 Limitations & Future Work

We describe several limitations in the current scope of the study and identify avenues for future work to build upon our findings:

First, the utility of the CodeA11y plugin was limited by the constraints placed by our target development environment (Visual Studio Code). Because CodeA11y was implemented as a Copilot plugin, we could only access a few APIs available to standard VSCode IDE plugins. The Copilot plugin infrastructure was limiting because it restricted the source code that could be passed to the model (*i.e.*, context window length). Our implementation contained some mechanisms for heuristically determining the most relevant files but ultimately serves as a proof of concept of what would be possible in the future, well-integrated version (*e.g.*, built into Copilot). These factors affected the code generation of our system.

Second, although our user study provides statistical evidence that CodeA11y helps developers write more accessible website code, we acknowledge certain limitations of AI coding assistants that could affect their overall effectiveness and reliability. One well-documented issue, particularly with proactive AI assistants [7], is their potential to provide untimely or irrelevant guidance. For instance, the models may occasionally suggest fixes for problems that do not exist (*i.e.*, false positives), such as recommending changes to code that is already fully compliant with accessibility standards. While our study did not surface such occurrences – likely because CodeA11y’s suggestions were tied directly to verified issues from an accessibility

checker, rather than the tool identifying issues on its own – this risk becomes more salient as AI assistants evolve to more proactively identify and address accessibility problems. Still, prior research suggests that developers often tolerate false positives more readily than false negatives [89], reasoning that overly cautious guidance from an assistant is less harmful than failing to flag genuine accessibility issues. Indeed, even standalone accessibility checkers, which are also known for producing false positives [48], have been widely adopted due to their overall beneficial effect on UI quality. Moreover, the occasional presence of false positives does not necessarily negate the value of employing such tools. By raising awareness and prompting developers to consider accessibility from the outset, AI assistants can help cultivate a proactive mindset toward inclusive design. In this sense, the technology does not need to achieve perfect accuracy to have a net positive effect. As the underlying models and APIs improve, and assistants become better integrated with real-world workflows, their precision and utility in improving accessibility are likely to increase.

Third, while our study demonstrates the potential of CodeA11y to encourage developers to adopt accessibility practices, we acknowledge that the broader impact of AI coding assistants on long-term learning and behavior changes (e.g., for accessibility awareness) remains underexplored in the current scope of the study. Research on how real-time AI tools can help developers internalize new practices, such as accessibility, or foster long-term behavioral changes could have strengthened the case for CodeA11y’s instructional components. For instance, prior work has shown that AI coding tools can enhance immediate task performance but may not consistently lead to deeper learning or sustained skill retention [49]. Similarly, studies on meta-cognitive demands in AI-assisted workflows emphasize the importance of tools promoting reflective learning and adaptive strategies, particularly as developers integrate them into their daily practices [37]. Although our findings suggest that CodeA11y has the potential to raise awareness of accessibility issues through direct integration with verified accessibility checks, further research is needed to understand whether such tools can foster a lasting developer’s commitment to accessibility or similar best practices. Additionally, exploring how these tools impact broader developer workflows, collaboration habits, and the ability to generalize learned behaviors across contexts would provide a more comprehensive view of their instructional value. By examining these dimensions, future studies could better elucidate the role of AI coding assistants in shaping not just productivity, but also the culture of inclusive and responsible software development. While CodeA11y focuses on improving accessibility, its approach could extend to other non-functional requirements, such as privacy and security. Investigating how specialized copilots could be seamlessly invoked within mainstream coding assistants for high-stakes scenarios – such as leveraging CodeA11y for front-end development tasks – represents a promising direction.

Finally, we see opportunities to iterate on and refine CodeA11y’s design. Our conservative approach adhered closely to Copilot’s existing interface to minimize friction during adoption. Future work could explore how developers respond to new features and interactions, identifying areas where innovation could enhance usability and functionality without compromising adoption. By addressing current limitations and exploring broader applications, tools like CodeA11y can refine how developers approach accessibility and other critical non-functional requirements. Beyond technical improvements, such advancements hold the potential to redefine AI’s role in shaping more inclusive, secure, and efficient coding practices.

2.10 Conclusion

This work bridges decades of accessibility efforts with AI coding assistants, offering a novel solution to persistent web accessibility challenges. Through a formative user study, we identify shortcomings in how current AI-assisted development workflows handle accessibility implementation. Accordingly, we develop CodeA11y, a GitHub Copilot Extension, and demonstrate that novice developers using it are significantly more likely to create accessible interfaces. By focusing on integrating accessibility improvements seamlessly into everyday development workflows, this work marks a first step toward fostering accessibility-conscious practices in human-AI collaborative UI development.

On *Preserving* Accessibility Metadata

Chapter 3

What You See Is What You Mean PDF Tagging

Most academic research is ultimately disseminated through documents in the PDF format. This format has advantages in flexibility and portability, but presents challenges for accessibility that have stubbornly resisted solutions despite decades of attempts. Tagging PDFs is hard to automate because tags are currently generated visually, not semantically, which makes the output cluttered and manual correction tedious and error-prone. Ironically, this semantic structure already exists during authoring, especially in What You See Is What You Mean (WYSIWYM) authoring tools like LaTeX, but is discarded during PDF rendering. This raises an obvious question, can we use this lost semantic information to better automate tagging in PDFs? In this project, we refine generated metadata using the semantics in the source documents of research papers. We demonstrate that the metadata generated by our method already surpasses what authors currently submit to ACM ASSETS on many criteria. Our approach represents a concrete step toward finally automating tagging and reading order in PDFs, directly improving the accessibility of our academic research.

3.1 Preamble

The formative work presented in this chapter is based on:

- [1] Peya Mowar, Aaron Steinfeld, and Jeffrey P Bigham. “We Write Our Research Papers in WYSIWYM. Why Do We Tag Our PDFs in WYSIWYG?”. In: *27th International ACM SIGACCESS Conference on Computers and Accessibility*. To appear. 2025

3.2 Introduction

Most academic research is disseminated through documents in the Portable Document Format (PDF), primarily because this format preserves formatting across many different platforms. However, persistent challenges remain in ensuring that documents prepared in this format consistently include the metadata necessary to make them accessible to people with disabilities [103, 100]. The PDF format is used almost exclusively in the academic community (even at ACM ASSETS), despite its shortcomings in accessibility.

Making a PDF accessible requires three primary kinds of metadata, which directly translate to how assistive technology conveys the content: *(i)* tagging individual sections of the PDF content according to their type and location on the page (*e.g.*, heading, paragraph, figure, etc.), *(ii)* defining an appropriate reading order, and *(iii)* providing content-specific metadata (*e.g.*, alternative text description for images or structured information for tables and lists). In this work, we focus on the first two categories of metadata (tags and reading order), primarily because these are most tractable to be “solved” by machine learning and yet even PDFs labeled by authors still show significant problems on these dimensions.

The tools available for providing this necessary metadata (called accessibility remediation) are tedious to use and it is difficult to know if you have made the PDF accessible correctly. Although numerous tools exist for PDF remediation [5, 41, 1], they work in the same way – the tool visualizes each page of the PDF document including any existing metadata information and the user must manually provide (or fix) the metadata. Because the metadata is only visible via specialized remediation tools and assistive technologies, it often goes unchecked or even accidentally removed during various processing steps during publication.

Given the general difficulty of working with PDFs, several projects have instead approached the problem as one of transforming the PDF into formats that are easier to work with, such as HTML. As an example, one goal of the ACM TAPS¹ process is to gather sufficient data from authors in order to produce papers in multiple formats, including both PDF and HTML. Despite extensive work in preparing source documents to comply with TAPS, authors still need to manually remediate their PDFs for them to be accessible. HTML versions are created as part of the ACM Digital Library, but they tend not to be as easily available or shared, such as on authors’ homepages.

In this work, we leverage the multiple representations of papers that are naturally created during the authoring process to automatically obtain this metadata. In particular, we use both the source document of the paper (LaTeX) and its visual rendering (PDF) to create highly accurate tags in the correct reading order. First, we run an object detection model on

¹<https://authors.acm.org/proceedings/production-information/taps-production-workflow>

the visual rendering, and an optical character recognition on each segment to determine what text it contains. We then use the text in each segment as an index into the source document, which we use to find related metadata (*e.g.*, section heading levels), and to influence the reading order where visual layout is unclear (common in two column formats).

We believe our method is starting to approach the quality level needed to perform tagging and reading order remediation automatically. While we advise being appropriately cautious, we are not the first to recognize that AI may be starting to get to the point where it may obviate the need for human intervention and that this may allow us to finally solve long-standing problems in accessibility [38]. In the particular instance of PDF tagging and reading order, we believe this is likely to become nearly solved because the information necessary to do this well is already available when producing PDFs, it just exists in disparate sources.

3.3 Related Work

Making academic paper PDFs accessible has remained a persistent challenge over the past decade [103, 20]. This is partly due to the human challenges in making research papers accessible, and partly due to the poor usability of existing PDF remediation tools. We discuss these challenges and the state of current approaches to PDF remediation.

3.3.1 Why Do Our Research Papers Remain Inaccessible?

As a research community, we have adopted PDF as the standard format for publishing due to its consistency and portability. However, this choice significantly undermines accessibility. PDFs are rooted in visual formatting, which makes it difficult to extract their underlying structure. As a result, despite the availability of many automated remediation tools [41, 1, 5], these often generate incorrect tag hierarchies that require manual review and correction. Even newer tools intended to simplify the process come with a steep learning curve [5]. Because these human-in-the-loop workflows still rely on visual tagging, they remain inaccessible to blind authors [64]. Yet authors are expected to use these limited (and often expensive!) tools to navigate a complex and time-consuming remediation process [64], making it difficult for them to do a good job (if at all).

Prior work explored various strategies including requesting, mandating, or distributing the responsibility among authors and other stakeholders in the publishing process (*e.g.*, publishers, volunteers, paid agencies) [100]. Still, tired of waiting for better tools, they found getting rid of the PDF format was the only viable solution. Consequently, SciA11y [77] and Paper to HTML [58] convert PDFs, while ar5iv [13] and the ACM Digital Library convert

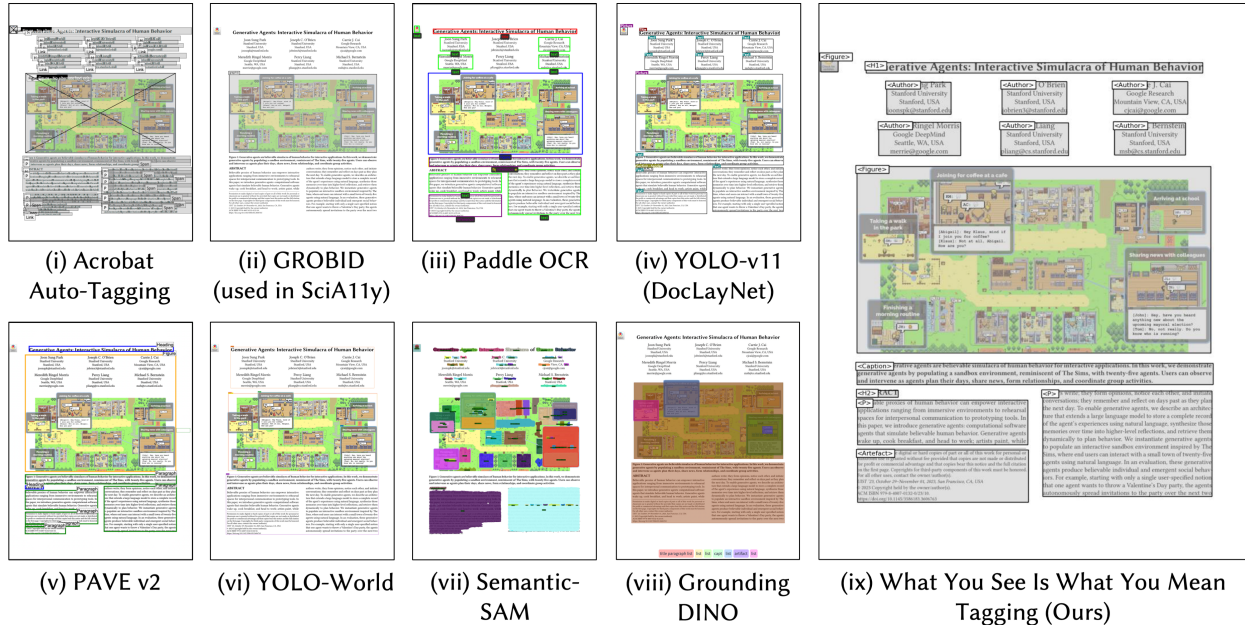


Figure 3.1: Visualizations of metadata generated by (a) accessibility remediation tools (i, v), (b) machine learning pipelines for document layout analysis (ii–iv), (c) vision-language models for object detection (vi–viii), and (d) our tagging approach (ix). Existing approaches rely solely on visual features of the PDF, *i.e.*, adopting a “What You See Is What You Get” paradigm, producing cluttered, incomplete or incorrect metadata. In contrast, our “What You See Is What You Mean” tags are refined using additional context from source documents (e.g., LaTeX files), resulting in cleaner and more semantically accurate results.

source documents, into an accessible HTML format. Yet, the uptake of alternative formats in scholarly communication remains limited.

Given these challenges in retroactively fixing PDFs, the LaTeX tagged PDF [28] project aims to automatically produce tagged PDFs directly from LaTeX source files. While this is a promising long-term solution, the authors document years of intensive engineering effort to make it viable. In contrast, we argue that source files can be leveraged for PDF remediation today, and our work is the first to explore this approach.

3.3.2 Can Vision-Language Models Help?

Recent advances in document understanding rely heavily on machine learning pipelines (including large language models) that attempt to parse layout, extract text, and even answer questions (e.g., DocVQA [72]). These tasks are useful for information retrieval and limited forms of semantic linking, and they are sometimes used downstream in systems that convert PDFs to alternative formats (for example, both SciA11y and Paper to HTML use the GROBID ML pipeline [112]). However, these tasks remain narrow in scope and are unable to

recover the precise spatial structure or order required to visually tag PDFs themselves.

Object Detection models, such as YOLO [101], trained or fine-tuned on document layout datasets (*e.g.*, PubLayNet [93], DocLayNet [67] or DocBank [84]) can often detect visual elements on pages like text and figures with high accuracy. However, they struggle to distinguish finer semantic categories like paragraphs and captions. Moreover, they are limited to a fixed set of labels and often overlook elements important for accessibility, such as background artifacts. In theory, vision-language models (VLMs) like GPT-4V for object detection could overcome these limitations by using free-form text prompts to infer more nuanced structure. In practice, however, most existing approaches [8, 44, 22] perform poorly on document tagging tasks (see Figure 3.1), likely due to the scarcity of semantically tagged documents in their training sets.

Still, the notion of providing “context” offers a promising direction. Our key insight is that source documents can supply the missing semantic context needed for PDF tagging in these foundation models. In our work, we explore an in-context learning approach to map the source documents to the PDFs that leverages source documents to guide and refine visual tags in their corresponding PDFs.

3.4 Methodology

Our approach (see Figure 3.2) processes the PDF visually to generate content regions, refines them by taking additional semantic context from the source documents, and ultimately produces content tags and a coherent reading order.

3.4.1 Visual Rendering (PDF) Processing

Identifying Content Regions

We use the Ultralytics YOLOv11 model [19] to identify content regions and their respective types in page images extracted from PDFs. We perform inference on a fine-tuned version² of the model on the DocLayNet dataset [66] to obtain content region bounding boxes, labels, and confidence scores for each page of the PDF.

Extracting Text from Regions

We extract the text contained on the page using PyTesseract v0.3.13³, a wrapper for Google’s Tesseract OCR Engine [122]. PyTesseract returns word-level bounding boxes for all detected

²<https://github.com/moured/YOLOv11-Document-Layout-Analysis>

³<https://pypi.org/project/pytesseract/>

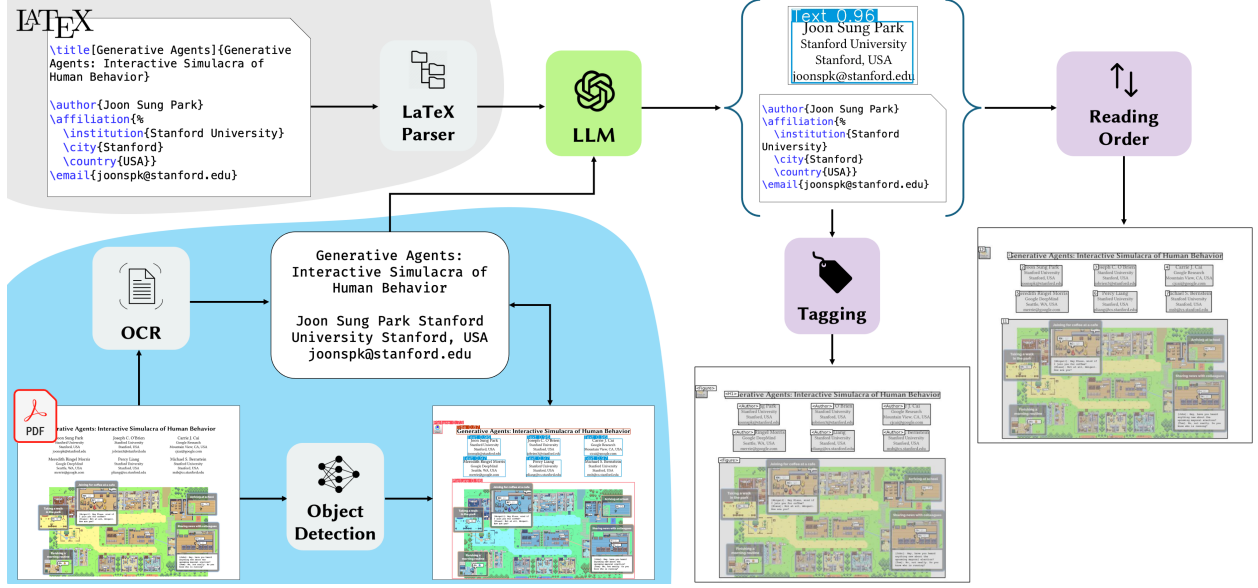


Figure 3.2: Our proposed method comprises of four steps: (i) processing the visual rendering of PDFs (in blue background), (ii) parsing the source document (in gray background), (iii) aligning the visual and structural representations using an LLM, and (iv) generating the metadata (tags and reading order).

text on the page, along with the corresponding recognized text. We map the word-level bounding boxes to the corresponding predicted content region, and aggregate the recognized text within each region. If textual content exists outside the current set of content regions, we treat it as standalone and add it as individual content regions labeled with `text`.

3.4.2 Source Document Parsing

Linearizing the Semantic Source

The input source document(s) are linearized to form a single semantic context for further remediation of the generated content regions. Our parser processes the `main.tex` file and recursively incorporates content from all included files (e.g., `<section>.tex` and `main.bbl`).

Chunking the Semantic Context

We pre-process the semantic context to remove comments, package imports, and other non-visual metadata. The cleaned document is then segmented into “chunks” – snippets of LaTeX code that are expected to render distinct visual content regions on a page. These chunks may include elements such as the title (`\title{}`), authors (`\author{}`), section headings (`\section{}` or `\subsubsection{}`), lists `\begin{itemize}` or `\begin{enumerate}`), refer-

ences (`@article{}`), footnotes (`\footnote{}`), and other meaningful constructs. We use hand-crafted regular expressions to split the source context.

3.4.3 Aligning Visual and Source Representations

Mapping Visual Content to Semantic Chunks

We align the OCR text output with the corresponding semantic chunk for each content region. First, we sort and filter the top 10 semantic chunks based on their forward and reverse word overlap scores with the OCR content. Let T be the multi-set of words in the OCR text, and S be the multi-set of words in the semantic snippet, then these scores are computed as:

$$\text{Forward_Overlap}(S, T) = \frac{|S \cap T|}{|T|}, \quad \text{Reverse_Overlap}(S, T) = \frac{|S \cap T|}{|S|} \quad (3.1)$$

Considering both these scores ensures the semantic chunk sufficiently explains the OCR text (forward), and that the OCR text meaningfully reflects the snippet chunk (reverse). This list is finally filtered by prompting GPT-4o, a Large Language Model (LLM). The prompt instruction highlights given to the LLM agent are:

Task: “You are given OCR text from a rendered LaTeX document and a list of LaTeX source chunks. Your task is to identify which specific LaTeX chunk(s) most likely rendered the visual content from the OCR output.”

Input: OCR Text: (`{ocr_text}`), LaTeX Chunks (with IDs): [`{chunk_snippet}`, ...]

Output: JSON ordered list of chunk identifiers, e.g., [`"chunk_001"`, `"chunk_011"`, ...]

Instructions:

- Identify the smallest set of chunks that best explain the OCR text. Each selected chunk must contribute content not already covered by earlier ones. Do not include multiple chunks that contain the same content.
- It is acceptable for a chunk to contain extra content, as long as part of it matches the OCR. Matching does not need to be exact. (E.g., `\begin{abstract}`) may also render the ABSTRACT heading. However, the OCR text needs to reasonably be present in the code content.
- If the OCR text contains metadata likely from background elements (e.g., asides, repetitive headers and footers), such as the ACM reference format, submission dates, copyright information, or if none of the LaTeX chunks plausibly match the OCR text, return an empty list: []

Refining Mappings and Regions

The LLM agent returns an initial mapping of semantic chunks to content regions, identifying which snippets most plausibly render each visual element. We further refine this mapping through two adjustments:

- **Splitting or Merging Content Regions:** When one content region maps to multiple semantic chunks, distinct elements (*e.g.*, a section and a subsection header) have been erroneously merged into one content region. Conversely, when multiple content regions correspond to a single semantic chunk, a coherent visual element, such as a paragraph or list, has been incorrectly fragmented into several bounding boxes. We use bounding box area heuristics to correct such cases.
- **Mapping Figures to Semantic Chunks:** Since our mapping is based on text overlap, it does not associate visual elements with their corresponding semantic chunks (*i.e.*, `\begin{figure}` blocks). We identify content regions labeled as `Picture` in DocLayNet taxonomy and associate them with the nearest text region that has already been assigned to a figure chunk (typically the caption). We use a visual proximity heuristic (threshold: 4% page height) to determine this mapping.

3.4.4 Metadata Generation

Generating Content Tags

After obtaining the mapped semantic chunk for each content region, we use a combination of regular expressions against the semantic chunk to correct or refine the visual label and generate the appropriate content tags. A content region not mapped to any semantic chunk is treated as an `<Artifact>`. Our current implementation produces 13 WYSIWYM tags: `<H1:H3>`, `<Paragraph>`, `<Caption>`, `<Table>`, `<List>`, `<Figure>`, `<Artifact>`, `<Footnote>`, `<Formula>`, `<BibEntry>`, and `<Author>`. While our content region detector (Section 3.4.1) is trained on the DocLayNet taxonomy, our method produces more granular and semantically meaningful tags, often also correcting classification errors in the process.

Identifying Reading Order

We use the semantic chunk associated with each content region as an anchor for recovering the intended reading order of the PDF. Specifically, we sort content regions on each page based on the position of their corresponding semantic chunk in the semantic context file. This ensures the reading order follows the author intent: for example, reading footnotes at

their mention, paragraphs spanning columns uninterrupted, associating figures with their captions, and skipping artifacts.

3.5 Experiments

We performed a manual evaluation of our proposed system by assembling a dataset of ACM research papers along with their source files. We compare our approach with two baselines: (i) Adobe Acrobat Pro Auto-Tagging, the most widely used PDF remediation software, and (ii) the high standard of author-submitted PDFs to ASSETS [98].

3.5.1 Dataset

We targeted papers published at the ACM CHI and ASSETS conferences over the past six years, as these venues require PDF accessibility tagging and recent papers are likely prepared with the latest remediation tools. We searched for them on arXiv and filtered for entries including source documents. For each paper, we downloaded the LaTeX source from arXiv and the corresponding PDF from the ACM Digital Library to obtain the authors' final submitted, tagged versions, discarding any entries where the two differ. In total, we collected 40 research paper PDFs (20 from CHI; 20 from ASSETS), comprising 554 pages (257 ASSETS; 297 CHI) and over 10,000 visual segments that required tagging.

3.5.2 Metadata Quality Metrics

To evaluate the quality of automatically generated tags and their reading order for accessibility, we defined metrics tailored to the specific requirements of PDF accessibility. Traditional ML metrics for object detection, such as Intersection over Union (IoU) or mean Average Precision (mAP), are not directly aligned with the goals of content localization and semantic tagging for two reasons: (i) we lack precise ground truth annotations for these tasks, and (ii) there are multiple valid interpretations of how content may be tagged. For example, a visual region may be represented by a single bounding box or as several smaller boxes, depending on the tagging granularity. Similarly, multiple reading sequences may be valid depending on the content structure. Instead, we focused on defining error metrics that can be computed manually:

1. To determine the how well content on a page is localized, we define Box Error Rate analogous to Word Error Rate (WER):

$$BER = \frac{(I + D + S)}{N} \quad (3.2)$$

where I is the number of insertions (missed content with no bounding box), D is the number of deletions (extra or redundant boxes), S is the number of substitutions (bounding boxes that do not fully contain their intended content), and N is the total number of expected content regions. Assuming our method generates T tags, we compute the reference count as $N = T + I - D$. The values of I , D , and S were counted manually for each page. Then, bounding box accuracy can be computed as $\text{Accuracy} = 1 - \text{BER}$.

2. To evaluate tag classification, we define Classification Error Rate (CER) as:

$$\text{CER} = \frac{\text{Number of incorrectly classified tags (C)}}{\text{Total number of generated tags (T)}} \quad (3.3)$$

We count C by checking for classification mismatches manually. Redundant tags (D) should be labeled as artifact, else they are considered misclassified. The precision of the predicted tag labels can then be computed as $\text{Precision} = 1 - \text{CER}$.

3. Finally, we evaluate reading order by computing Reading Order Error Rate (ROER) as:

$$\text{ROER} = \frac{\text{Number of misplaced tags (M)}}{\text{Total number of generated tags (T)}} \quad (3.4)$$

If a tag does not follow the natural reading order, it is manually counted as misplaced. Similar to CER, we can compute reading order precision as $\text{Precision} = 1 - \text{ROER}$.

3.5.3 Accessibility Criteria

We evaluated how well our generated metadata fulfills PDF accessibility criteria, as defined in prior work [5]. Based on our collected dataset and predicted tags, we selected a subset of the recommended criteria: tagging of all content, headings, heading levels, figures, tables, lists, captions, and correct reading order. More granular tagging, such as tagging list items and table cells, is left for future work. We manually evaluated a subset of our dataset (10 random ASSETS papers) against the versions submitted by the authors.

3.6 Results

3.6.1 Quantitative Results

Overall, our method demonstrated high performance across all three tasks (content localization, semantic tagging and reading order identification) on our collected dataset, as shown in

Table 3.1. Our method also surpassed authors’ submitted manually tagged PDF metadata on several accessibility criteria (depicted in Table 3.2): reading order, tagging tables and captions. We further analyzed our predicted tags to identify failure cases in our approach. Figure 3.3 illustrates a bar graph showing the frequency of tag misclassifications across the dataset. Occasionally, our word overlap-based heuristics fail to identify corresponding semantic chunks (most commonly for headings), resulting in incorrect tagging as `<Artifact>`.

| Criteria | Error Rate [%] | Accuracy [%] | Precision [%] |
|----------------|----------------|--------------|---------------|
| Bounding Boxes | 4.72 | 95.28 | – |
| Classification | 4.03 | – | 95.96 |
| Reading Order | 2.74 | – | 97.26 |

Table 3.1: Metadata error analysis from a manual evaluation of 40 randomly selected papers from ACM CHI and ASSETS (2019-2024).

| Criteria | Adobe Acrobat Auto-Tags [%] | What Authors Submitted [%] | WYSIWYM Tags (Ours) [%] |
|-------------------------|--------------------------------|-------------------------------|----------------------------|
| All Content Tagged | 100.0 | 100.0 | 100.0 |
| Reading Order | 89.66 | 95.42 | 96.26 |
| Headings Tagged | 90.47 | 94.78 | 82.68 |
| Headings Tagged + Level | 85.28 | 94.35 | 81.82 |
| Tables Tagged | 68.18 | 50.0 | 95.0 |
| Lists Tagged | 60.34 | 85.36 | 75.61 |
| Figures Tagged | 52.24 | 100.0 | 84.0 |
| Captions Tagged | 0.0 | 6.45 | 95.65 |
| Average Score | 71.79 | 78.21 | 88.75 |

Table 3.2: Comparison of PDF accessibility tagging between metadata generated by Acrobat’s auto-tagging feature, what authors submitted and our generated metadata. To compare the results, we manually evaluate 10 randomly selected papers from ACM ASSETS (2019-2024).

3.6.2 Qualitative Results

Figures 3.4 and 3.5 demonstrate positive and negative examples of our generated metadata outputs respectively. Our system correctly identifies notes and captions (Figure 3.4), which were often incorrectly tagged as `<Paragraph>` by both Acrobat and the authors. However, our current implementation cannot process content in other languages (see Figure 3.5(a)). Moreover, our refinements in Section 3.4.3 were not always able to correct content region

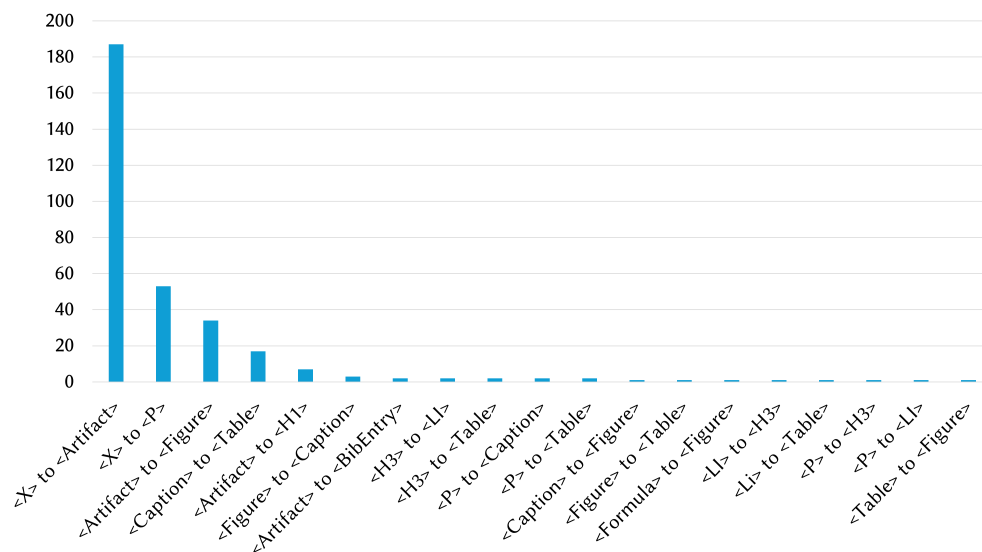


Figure 3.3: Classification error frequencies across 40 PDFs comprising 554 pages ($N = 10,261$ tags). Each bar represents the frequency of misclassification from a ground truth tag to a predicted tag (denoted as `<True>` to `<Predicted>`). The most common error involves classifying various tags as `<Artifact>`, indicated by `<X>` to `<Artifact>`.

boundaries, especially when dealing with non-textual content, such as figures or sub-figures (see Figure 3.5(b)).

3.6.3 Limitations

Our current implementation of this work presents several limitations which suggest avenues for future work. First, our method relies on heuristics to map and refine multiple representations, which struggle to generalize in observed failure cases. Future work could focus on building a large accessibility metadata corpus to support the training of auto-tagging approaches. Second, our approach can be refined to generate richer semantic metadata, including alt-texts for figures and equations and structured tagging for table and list elements, from the source documents. For example, we could further chunk the `\itemize` command into list items defined by `\item`. Third, while we demonstrate our approach works for LaTeX source documents, it can also be extended for WYSIWYG authoring tools like MS Word, which preserve a linear reading order. Finally, our work does not endorse the continued dominance of PDFs in academic publishing; formats like HTML and ePub offer far better accessibility support. However, given its pervasiveness in current research dissemination practices, we believe improving accessibility through better authoring and PDF remediation tools remains necessary, even as the community moves toward more accessible standards.

3. What You See Is What You Mean PDF Tagging

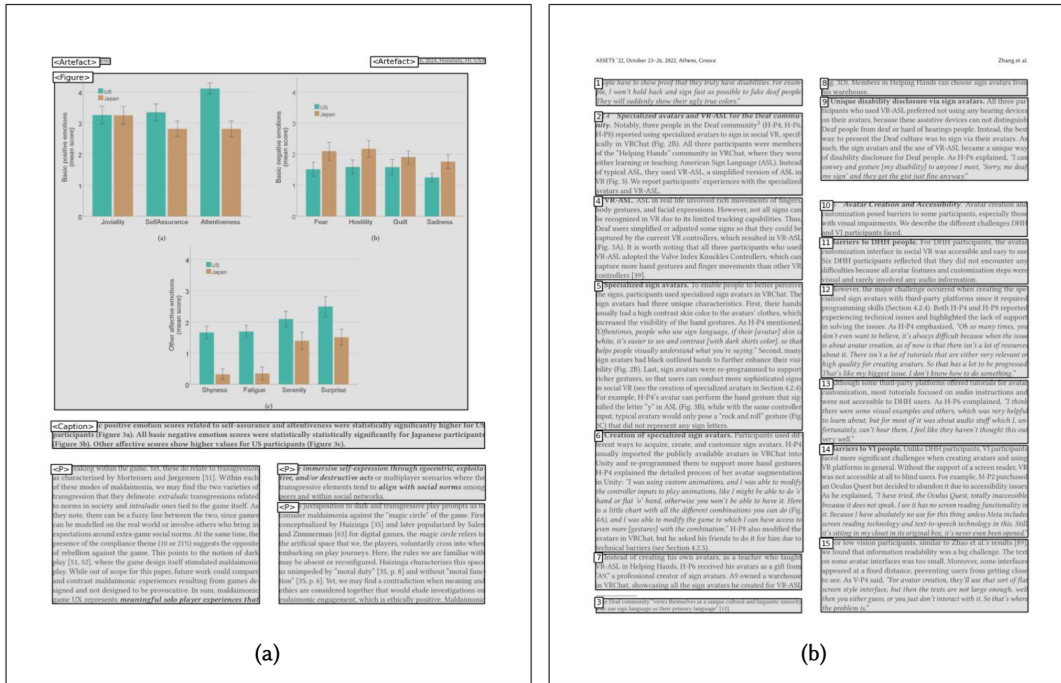


Figure 3.4: Positive examples from our method: (a) Our approach accurately classifies figures, captions, and artifacts; (b) It correctly predicts footnote order while skipping artifacts.

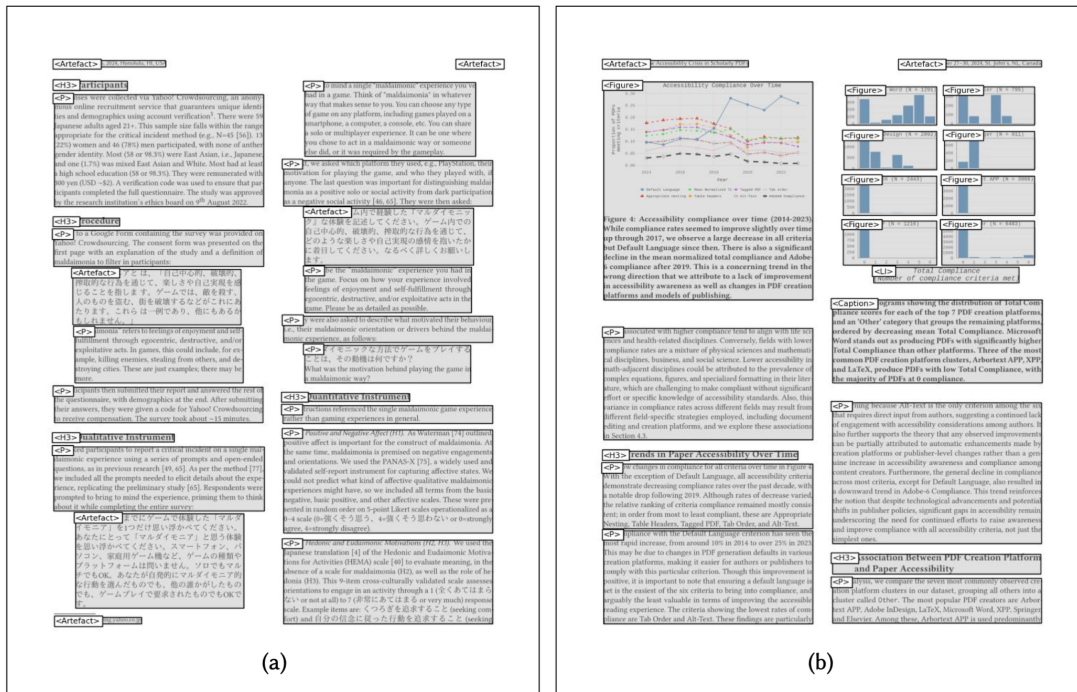


Figure 3.5: Negative examples from our method: (a) Paragraphs in Japanese are tagged artifacts; (b) Figures are occasionally merged with captions or split into sub-figures.

3.7 Conclusion

In this chapter, we introduce a novel approach that combines visual tagging of PDFs with semantic cues from the source document, enabling more accurate and robust metadata generation. Our method offers what we believe to be the most reliable solution to date for automated PDF tagging and surpasses the quality of manual tagging provided by authors in our collected dataset. This work represents a concrete first step toward the long-standing goal of accurately and automatically tagging PDF documents.

Chapter 4

Conclusion

This thesis serves as a stepping stone for automating accessibility directly in existing digital authoring workflows. It presents two AI tools that address the metadata flow problem in web and PDF accessibility, and demonstrates their effectiveness through comprehensive user and manual evaluations.

4.1 Key Contributions

The key contributions of this work are:

- A study with 16 developers that uncovered both benefits and limitations of current AI coding assistants for authoring accessible UI code.
- CodeA11y¹: a GitHub Copilot Extension that generates accessible UI code, identifies existing issues and reminds developers to perform manual validation.
- A dataset and associated metrics specifically targeted at what matters for accessibility of PDF research papers.
- A novel method combining visual and source representations of PDF documents to improve detection of tags and reading order, along with an evaluation establishing a strong baseline for this dataset.

¹The source code for CodeA11y is available at <https://github.com/peyajm29/codea11y/>.

Bibliography

- [1] Equidox Software Company. *Equidox PDF Accessibility Software*. <https://equidox.co/pdf-solutions/pdf-accessibility-software/>. Accessed: 2025-04-17. 2025.
- [2] Peya Mowar, Aaron Steinfeld, and Jeffrey P Bigham. “We Write Our Research Papers in WYSIWYM. Why Do We Tag Our PDFs in WYSIWYG?” In: *27th International ACM SIGACCESS Conference on Computers and Accessibility*. To appear. 2025.
- [3] Peya Mowar et al. “CodeA11y: Making AI Coding Assistants Useful for Accessible Web Development”. In: *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 2025, pp. 1–15.
- [4] Yi-Hao Peng et al. “DreamStruct: Understanding Slides and User Interfaces via Synthetic Data Generation”. In: *European Conference on Computer Vision*. Springer. 2025, pp. 466–485.
- [5] Felix M. Schmitt-Koopmann, Elaine Huang, and Alireza Hutter Hans-Peter Darvishy. “Towards More Accessible Scientific PDFs for People with Visual Impairments: Step-by-Step PDF Remediation to Improve Tag Accuracy”. In: *Proceedings of the ACM Conference on Human Factors and Computing* (2025).
- [6] Wajdi Aljedaani et al. “Does ChatGPT Generate Accessible Code? Investigating Accessibility Challenges in LLM-Generated Source Code”. In: *Proceedings of the 21st International Web for All Conference*. W4A '24. Singapore, Singapore: Association for Computing Machinery, 2024, pp. 165–176. ISBN: 9798400710308. DOI: [10.1145/3677846.3677854](https://doi.org/10.1145/3677846.3677854). URL: <https://doi.org/10.1145/3677846.3677854>.
- [7] Valerie Chen et al. “Need Help? Designing Proactive AI Assistants for Programming”. In: *arXiv preprint arXiv:2410.04596* (2024).
- [8] Tianheng Cheng et al. “Yolo-world: Real-time open-vocabulary object detection”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2024, pp. 16901–16911.
- [9] Giovanni Delnevo, Manuel Andruccioli, and Silvia Mirri. “On the Interaction with Large Language Models for Web Accessibility: Implications and Challenges”. In: *2024 IEEE 21st Consumer Communications & Networking Conference (CCNC)*. IEEE. 2024, pp. 1–6.
- [10] Deque Systems. *axe: Accessibility Testing Tools and Software*. Accessed: 2024-12-10. 2024. URL: <https://www.deque.com/axe/>.
- [11] dkandalov. *activity-tracker*. <https://github.com/dkandalov/activity-tracker>. 2024.

BIBLIOGRAPHY

- [12] Abhimanyu Dubey et al. “The llama 3 herd of models”. In: *arXiv preprint arXiv:2407.21783* (2024).
- [13] Charles Frankston et al. “HTML papers on arXiv—why it is important, and how we made it happen”. In: *arXiv preprint arXiv:2402.08954* (2024).
- [14] *GitHub Copilot*. <https://github.com/features/copilot>. [Accessed: 2024-04-22]. GitHub, 2024.
- [15] Google. *Chrome DevTools Documentation*. Accessed: 2024-12-10. 2024. URL: <https://developer.chrome.com/docs/devtools>.
- [16] Binyuan Hui et al. “Qwen2. 5-coder technical report”. In: *arXiv preprint arXiv:2409.12186* (2024).
- [17] *IAB Website Categories*. <https://docs.webshrinker.com/v3/iab-website-categories.html#iab-categories>. Accessed: 2024-04-22. 2024.
- [18] IBM. *IBM Equal Access Toolkit*. Accessed: 2024-12-10. 2024. URL: <https://www.ibm.com/able/toolkit>.
- [19] Glenn Jocher and Jing Qiu. *Ultralytics YOLO11*. Version 11.0.0. 2024. URL: <https://github.com/ultralytics/ultralytics>.
- [20] Anukriti Kumar and Lucy Lu Wang. “Uncovering the New Accessibility Crisis in Scholarly PDFs”. In: *arXiv preprint arXiv:2410.03022* (2024).
- [21] Haotian Liu et al. “Visual instruction tuning”. In: *Advances in neural information processing systems* 36 (2024).
- [22] Shilong Liu et al. “Grounding dino: Marrying dino with grounded pre-training for open-set object detection”. In: *European conference on computer vision*. Springer, 2024, pp. 38–55.
- [23] Juan-Miguel López-Gil and Juanan Pereira. “Turning manual web accessibility success criteria into automatic: an LLM-based approach”. In: *Universal Access in the Information Society* (2024), pp. 1–16.
- [24] Anton Lozhkov et al. “Starcoder 2 and the stack v2: The next generation”. In: *arXiv preprint arXiv:2402.19173* (2024).
- [25] Beatriz Martins and Carlos Duarte. “Large-scale study of web accessibility metrics”. In: *Universal Access in the Information Society* 23.1 (2024), pp. 411–434.
- [26] Forough Mehralian et al. “Automated Code Fix Suggestions for Accessibility Issues in Mobile Apps”. In: *arXiv preprint arXiv:2408.03827* (2024).
- [27] Microsoft. *Accessibility Insights*. <https://accessibilityinsights.io>. Accessed: 2024-12-10. 2024.
- [28] Frank Mittelbach et al. “Automatically producing accessible and reusable PDFs with LATEX”. In: *Proceedings of the ACM Symposium on Document Engineering 2024*. 2024, pp. 1–4.
- [29] Peya Mowar. “Accessibility in AI-Assisted Web Development”. In: *Proceedings of the 21st International Web for All Conference*. 2024, pp. 123–125.
- [30] Peya Mowar et al. “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*. 2024, pp. 1–6.

- [31] Hussein Mozannar et al. “Reading between the lines: Modeling user behavior and costs in AI-assisted programming”. In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 2024, pp. 1–16.
- [32] BBC News. *BBC Home - Breaking News, World News, US News, Sports ...* 2024. URL: <https://www.bbc.com/>.
- [33] Kubernetes Project. *Kubernetes*. 2024. URL: <https://kubernetes.io/>.
- [34] *SimilarWeb - Website Traffic & Market Intelligence*. <http://similarweb.com>. Accessed: 2024-04-22. SimilarWeb, 2024.
- [35] Amanda Swearngin et al. “Towards Automated Accessibility Report Generation for Mobile Apps”. In: *ACM Transactions on Computer-Human Interaction* 31.4 (2024), pp. 1–44.
- [36] Maryam Taeb et al. “Axnav: Replaying accessibility tests from natural language”. In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 2024, pp. 1–16.
- [37] Lev Tankelevitch et al. “The metacognitive demands and opportunities of generative AI”. In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 2024, pp. 1–24.
- [38] Gregg Vanderheiden and Crystal Yvette Marte. “Will AI allow us to dispense with all or most accessibility regulations?” In: *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. CHI EA ’24. Honolulu, HI, USA: Association for Computing Machinery, 2024. ISBN: 9798400703317. DOI: [10.1145/3613905.3644059](https://doi.org/10.1145/3613905.3644059). URL: <https://doi.org/10.1145/3613905.3644059>.
- [39] WebAIM. *The WebAIM Million - The 2024 report on the accessibility of the top 1,000,000 home pages*. <https://webaim.org/projects/million/>. Accessed: 2024-04-22. 2024.
- [40] WebAIM. *The WebAIM Million - The 2025 report on the accessibility of the top 1,000,000 home pages*. <https://webaim.org/projects/million/>. Accessed: 2025-06-30. 2024.
- [41] Adobe Inc. *Adobe Acrobat Pro DC Accessibility Features*. <https://helpx.adobe.com/acrobat/using/create-verify-pdf-accessibility.html>. Adobe Systems Incorporated. 2023.
- [42] Jinze Bai et al. “Qwen technical report”. In: *arXiv preprint arXiv:2309.16609* (2023).
- [43] Jinze Bai et al. “Qwen-vl: A frontier large vision-language model with versatile abilities”. In: *arXiv preprint arXiv:2308.12966* (2023).
- [44] Jiaqi Chen, Zeyu Yang, and Li Zhang. *Semantic Segment Anything*. <https://github.com/fudan-zvg/Semantic-Segment-Anything>. 2023.
- [45] Danyang Fan et al. “The accessibility of data visualizations on the web for screen reader users: Practices and experiences during covid-19”. In: *ACM Transactions on Accessible Computing* 16.1 (2023), pp. 1–29.
- [46] Charlie Giattino et al. “Artificial Intelligence”. In: *Our World in Data* (2023). <https://ourworldindata.org/artificial-intelligence>.

BIBLIOGRAPHY

- [47] Mina Huh et al. “AVscript: Accessible Video Editing with Audio-Visual Scripts”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–17.
- [48] Syed Fatiul Huq et al. “# A11yDev: Understanding Contemporary Software Accessibility Practices from Twitter Conversations”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–18.
- [49] Majeed Kazemitabaar et al. “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*. 2023, pp. 1–23.
- [50] Amanda Li, Jason Wu, and Jeffrey P Bigham. “Using llms to customize the ui of webpages”. In: *Adjunct Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 2023, pp. 1–3.
- [51] Achraf Othman, Amira Dhouib, and Aljazi Nasser Al Jabor. “Fostering websites accessibility: A case study on the use of the Large Language Models ChatGPT for automatic remediation”. In: *Proceedings of the 16th International Conference on Pervasive Technologies Related to Assistive Environments*. 2023, pp. 707–713.
- [52] Luís P. Carvalho et al. “Towards real-time and large-scale web accessibility”. In: *Proceedings of the 25th International ACM SIGACCESS Conference on Computers and Accessibility*. 2023, pp. 1–9.
- [53] Maulishree Pandey and Tao Dong. “Blending Accessibility in UI Framework Documentation to Build Awareness”. In: *Proceedings of the 25th International ACM SIGACCESS Conference on Computers and Accessibility*. 2023, pp. 1–12.
- [54] Yi-Hao Peng et al. “Slide Gestalt: Automatic Structure Extraction in Slide Decks for Non-Visual Access”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–14.
- [55] Baptiste Roziere et al. “Code llama: Open foundation models for code”. In: *arXiv preprint arXiv:2308.12950* (2023).
- [56] Navid Salehnamadi, Ziyao He, and Sam Malek. “Assistive-Technology Aided Manual Accessibility Testing in Mobile Apps, Powered by Record-and-Replay”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI ’23. Hamburg, Germany: Association for Computing Machinery, 2023. ISBN: 9781450394215. DOI: [10 . 1145 / 3544548 . 3580679](https://doi.org/10.1145/3544548.3580679). URL: <https://doi.org/10.1145/3544548.3580679>.
- [57] Navid Salehnamadi, Forough Mehralian, and Sam Malek. “Groundhog: An Automated Accessibility Crawler for Mobile Apps”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’22. Rochester, MI, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: [10.1145/3551349.3556905](https://doi.org/10.1145/3551349.3556905). URL: <https://doi.org/10.1145/3551349.3556905>.
- [58] Lucy Lu Wang, Jonathan Bragg, and Daniel S Weld. “Paper to html: A publicly available web tool for converting scientific pdfs into accessible html”. In: *ACM SIGACCESS Accessibility and Computing* 134 (2023), pp. 1–1.
- [59] Jason Wu et al. “Webui: A dataset for enhancing visual ui understanding with web semantics”. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–14.

- [60] Yuxin Zhang et al. “Automated and Context-Aware Repair of Color-Related Accessibility Issues for Android Apps”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 1255–1267. ISBN: 9798400703270. DOI: [10.1145/3611643.3616329](https://doi.org/10.1145/3611643.3616329). URL: <https://doi.org/10.1145/3611643.3616329>.
- [61] Qinkai Zheng et al. “Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x”. In: *arXiv preprint arXiv:2303.17568* (2023).
- [62] Shakila Cherise S Joyner et al. “Visualization accessibility in the wild: Challenges faced by visualization designers”. In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 2022, pp. 1–19.
- [63] Lilu Martin et al. “The Landscape of Accessibility Skill Set in the Software Industry Positions”. In: *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility*. 2022, pp. 1–4.
- [64] Rachel Menzies, Garreth W Tigwell, and Michael Crabb. “Author reflections on creating accessible academic papers”. In: *ACM Transactions on Accessible Computing* 15.4 (2022), pp. 1–36.
- [65] Yi-Hao Peng et al. “Diffscriber: Describing Visual Design Changes to Support Mixed-Ability Collaborative Presentation Authoring”. In: *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 2022, pp. 1–13.
- [66] Birgit Pfitzmann et al. “DocLayNet: A Large Human-Annotated Dataset for Document-Layout Analysis”. In: (2022). DOI: [10.1145/3534678.353904](https://arxiv.org/abs/2206.01062). URL: <https://arxiv.org/abs/2206.01062>.
- [67] Birgit Pfitzmann et al. “Doclaynet: A large human-annotated dataset for document-layout segmentation”. In: *Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining*. 2022, pp. 3743–3751.
- [68] Athira Pillai, Kristen Shinohara, and Garreth W Tigwell. “Website builders still contribute to inaccessible web design”. In: *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility*. 2022, pp. 1–4.
- [69] Mark Chen et al. “Evaluating large language models trained on code”. In: *arXiv preprint arXiv:2107.03374* (2021).
- [70] Jaewook Lee et al. “Image Explorer: Multi-layered touch exploration to make images accessible”. In: *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. 2021, pp. 1–4.
- [71] Kelly Mack et al. “Designing tools for high-quality alt text authoring”. In: *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. 2021, pp. 1–14.
- [72] Minesh Mathew, Dimosthenis Karatzas, and CV Jawahar. “Docvqa: A dataset for vqa on document images”. In: *Proceedings of the IEEE/CVF winter conference on applications of computer vision*. 2021, pp. 2200–2209.
- [73] Joe McKendrick. “AI adoption skyrocketed over the last 18 months”. In: *Harvard Business Review* 27 (2021).

BIBLIOGRAPHY

- [74] Yi-Hao Peng, Jeffrey P Bigham, and Amy Pavel. “Slidecho: Flexible non-visual exploration of presentation videos”. In: *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. 2021, pp. 1–12.
- [75] Yi-Hao Peng et al. “Say it all: Feedback for improving non-visual presentation accessibility”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–12.
- [76] Navid Salehnamadi et al. “Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI ’21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: [10.1145/3411764.3445455](https://doi.org/10.1145/3411764.3445455). URL: <https://doi.org/10.1145/3411764.3445455>.
- [77] Lucy Lu Wang et al. “Improving the accessibility of scientific documents: Current state, user needs, and a system solution to enhance scientific PDF accessibility for blind and low vision users”. In: *arXiv preprint arXiv:2105.00076* (2021).
- [78] Jason Wu et al. “When can accessibility help? An exploration of accessibility feature recommendation on mobile devices”. In: *Proceedings of the 18th international web for all conference*. 2021, pp. 1–12.
- [79] Xiaoyi Zhang et al. “Screen recognition: Creating accessibility metadata for mobile applications from pixels”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–15.
- [80] Jieshan Chen et al. “Object detection for graphical user interface: Old fashioned or deep learning or a combination?” In: *proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1202–1214.
- [81] Jieshan Chen et al. “Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning”. In: *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 2020, pp. 322–334.
- [82] Cole Gleason et al. “Making GIFs Accessible”. In: *Proceedings of the 22nd International ACM SIGACCESS Conference on Computers and Accessibility*. 2020, pp. 1–10.
- [83] Cole Gleason et al. “Twitter A11y: A Browser Extension to Make Twitter Images Accessible”. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI ’20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–12. ISBN: 9781450367080. DOI: [10.1145/3313831.3376728](https://doi.org/10.1145/3313831.3376728). URL: <https://doi.org/10.1145/3313831.3376728>.
- [84] Minghao Li et al. “Docbank: A benchmark dataset for document layout analysis”. In: *arXiv preprint arXiv:2006.01038* (2020).
- [85] Rohan Patel et al. “Why Software is Not Accessible: Technology Professionals’ Perspectives and Challenges”. In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI EA ’20. Honolulu, HI, USA: Association for Computing Machinery, 2020, pp. 1–9. ISBN: 9781450368193. DOI: [10.1145/3334480.3383103](https://doi.org/10.1145/3334480.3383103). URL: <https://doi.org/10.1145/3334480.3383103>.
- [86] Amy Pavel, Gabriel Reyes, and Jeffrey P Bigham. “Rescribe: Authoring and automatically editing audio descriptions”. In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 2020, pp. 747–759.

- [87] Cole Gleason et al. ““It’s almost like they’re trying to hide it”: How user-provided image descriptions have failed to make Twitter accessible”. In: *The World Wide Web Conference*. 2019, pp. 549–559.
- [88] Cole Gleason et al. “Making memes accessible”. In: *Proceedings of the 21st International ACM SIGACCESS Conference on Computers and Accessibility*. 2019, pp. 367–376.
- [89] Rafal Kocielnik, Saleema Amershi, and Paul N Bennett. “Will you accept an imperfect ai? exploring designs for adjusting end-user expectations of ai systems”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–14.
- [90] Jingyi Li et al. “Editing spatial layouts through tactile templates for people with visual impairments”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–11.
- [91] Yi-Hao Peng et al. “Personaltouch: Improving touchscreen usability by personalizing accessibility settings based on individual user’s touchscreen interaction”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–11.
- [92] Venkatesh Potluri et al. “Ai-assisted ui design for blind and low-vision creators”. In: *the ASSETS’19 Workshop: AI Fairness for People with Disabilities*. 2019.
- [93] Xu Zhong, Jianbin Tang, and Antonio Jimeno Yepes. “Publaynet: largest dataset ever for document layout analysis”. In: *2019 International conference on document analysis and recognition (ICDAR)*. IEEE. 2019, pp. 1015–1022.
- [94] Darren Guinness, Edward Cutrell, and Meredith Ringel Morris. “Caption crawler: Enabling reusable alternative text descriptions using reverse image search”. In: *Proceedings of the 2018 chi conference on human factors in computing systems*. 2018, pp. 1–11.
- [95] Anne Spencer Ross et al. “Examining image-based button labeling for accessibility in Android apps through large-scale analysis”. In: *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*. 2018, pp. 119–130.
- [96] Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. “Robust annotation of mobile application interfaces in methods for accessibility repair and enhancement”. In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 2018, pp. 609–621.
- [97] Victoria Clarke and Virginia Braun. “Thematic analysis”. In: *The journal of positive psychology* 12.3 (2017), pp. 297–298.
- [98] Jonathan Lazar et al. “Making the field of computing more inclusive”. In: *Communications of the ACM* 60.3 (2017), pp. 50–59.
- [99] Xiaoyi Zhang et al. “Interaction proxies for runtime repair and enhancement of mobile application accessibility”. In: *Proceedings of the 2017 CHI conference on human factors in computing systems*. 2017, pp. 6024–6037.
- [100] Jeffrey P. Bigham et al. “An Uninteresting Tour Through Why Our Research Papers Aren’t Accessible”. In: *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. CHI EA ’16. San Jose, California, USA: Association for Computing Machinery, 2016, pp. 621–631. ISBN: 9781450340823. DOI: [10.1145/2851581.2892588](https://doi.org/10.1145/2851581.2892588). URL: <https://doi.org/10.1145/2851581.2892588>.

BIBLIOGRAPHY

- [101] Joseph Redmon et al. “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.
- [102] Katharina Reinecke, David R. Flatla, and Christopher Brooks. “Enabling Designers to Foresee Which Colors Users Cannot See”. In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. CHI ’16. San Jose, California, USA: Association for Computing Machinery, 2016, pp. 2693–2704. ISBN: 9781450333627. DOI: [10.1145/2858036.2858077](https://doi.org/10.1145/2858036.2858077). URL: <https://doi.org/10.1145/2858036.2858077>.
- [103] Erin Brady, Yu Zhong, and Jeffrey P Bigham. “Creating accessible PDFs for conference proceedings”. In: *Proceedings of the 12th International Web for All Conference*. 2015, pp. 1–4.
- [104] Yun Huang et al. “CAN: Composable accessibility infrastructure via data-driven crowdsourcing”. In: *Proceedings of the 12th International Web for All Conference*. 2015, pp. 1–10.
- [105] Jeffrey P Bigham. “Making the web easier to see with opportunistic accessibility improvement”. In: *Proceedings of the 27th annual ACM symposium on User interface software and technology*. 2014, pp. 117–122.
- [106] Markel Vigo, Justin Brown, and Vivienne Conway. “Benchmarking web accessibility evaluation tools: measuring the harm of sole reliance on automated tests”. In: *Proceedings of the 10th international cross-disciplinary conference on web accessibility*. 2013, pp. 1–10.
- [107] Yeliz Yesilada et al. “Understanding web accessibility and its drivers”. In: *Proceedings of the international cross-disciplinary conference on web accessibility*. 2012, pp. 1–9.
- [108] Jeffrey P Bigham, Jeremy T Brudvik, and Bernie Zhang. “Accessibility by demonstration: enabling end users to guide developers to web accessibility solutions”. In: *Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility*. 2010, pp. 35–42.
- [109] Morgan Dixon and James Fogarty. “Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2010, pp. 1525–1534.
- [110] Muhammad Asiful Islam, Yevgen Borodin, and I. V. Ramakrishnan. “Mixture model based label association techniques for web accessibility”. In: *Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology*. UIST ’10. New York, New York, USA: Association for Computing Machinery, 2010, pp. 67–76. ISBN: 9781450302715. DOI: [10.1145/1866029.1866041](https://doi.org/10.1145/1866029.1866041). URL: <https://doi.org/10.1145/1866029.1866041>.
- [111] Daisuke Sato et al. “Social accessibility: the challenge of improving web accessibility through collaboration”. In: *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*. 2010, pp. 1–2.
- [112] Patrice Lopez. “GROBID: Combining automatic bibliographic data recognition and term extraction for scholarship publications”. In: *International conference on theory and practice of digital libraries*. Springer. 2009, pp. 473–474.
- [113] Daisuke Sato et al. “What’s next? a visual editor for correcting reading order”. In: *Human-Computer Interaction–INTERACT 2009: 12th IFIP TC 13 International*

- Conference, Uppsala, Sweden, August 24-28, 2009, Proceedings, Part I 12*. Springer. 2009, pp. 364–377.
- [114] Hironobu Takagi et al. “Collaborative web accessibility improvement: challenges and possibilities”. In: *Proceedings of the 11th international ACM SIGACCESS conference on Computers and accessibility*. 2009, pp. 195–202.
 - [115] Jeremy T. Brudvik et al. “Hunting for headings: sighted labeling vs. automatic classification of headings”. In: *Proceedings of the 10th International ACM SIGACCESS Conference on Computers and Accessibility*. Assets '08. Halifax, Nova Scotia, Canada: Association for Computing Machinery, 2008, pp. 201–208. ISBN: 9781595939760. DOI: [10.1145/1414471.1414508](https://doi.org/10.1145/1414471.1414508). URL: <https://doi.org/10.1145/1414471.1414508>.
 - [116] Ben Caldwell et al. “Web content accessibility guidelines (WCAG) 2.0”. In: *WWW Consortium (W3C) 290.1-34* (2008), pp. 5–12.
 - [117] World Wide Web Consortium et al. “Web content accessibility guidelines (WCAG) 2.0”. In: (2008).
 - [118] André P Freire et al. “An evaluation of web accessibility metrics based on their attributes”. In: *Proceedings of the 26th annual ACM international conference on Design of communication*. 2008, pp. 73–80.
 - [119] Shinya Kawanaka et al. “Accessibility commons: a metadata infrastructure for web accessibility”. In: *Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility*. 2008, pp. 153–160.
 - [120] Jeffrey P Bigham and Richard E Ladner. “Accessmonkey: a collaborative scripting framework for web users and developers”. In: *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*. 2007, pp. 25–34.
 - [121] Giorgio Brajnik and Raffaella Lomuscio. “SAMBA: a semi-automatic method for measuring barriers of accessibility”. In: *Proceedings of the 9th International ACM SIGACCESS Conference on Computers and Accessibility*. 2007, pp. 43–50.
 - [122] Ray Smith. “An overview of the Tesseract OCR engine”. In: *Ninth international conference on document analysis and recognition (ICDAR 2007)*. Vol. 2. IEEE. 2007, pp. 629–633.
 - [123] Markel Vigo et al. “Quantitative metrics for measuring web accessibility”. In: *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*. 2007, pp. 99–107.
 - [124] Jeffrey P. Bigham et al. “WebInSight: making web images accessible”. In: *Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility*. Assets '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 181–188. ISBN: 1595932909. DOI: [10.1145/1168987.1169018](https://doi.org/10.1145/1168987.1169018). URL: <https://doi.org/10.1145/1168987.1169018>.
 - [125] David Sloan et al. “Contextual web accessibility-maximizing the benefit of accessibility guidelines”. In: *Proceedings of the 2006 international cross-disciplinary workshop on Web accessibility (W4A): Building the mobile web: rediscovering accessibility?* 2006, pp. 121–131.
 - [126] Jonathan Lazar, Alfreda Dudley-Sponaugle, and Kisha-Dawn Greenidge. “Improving web accessibility: a study of webmaster perceptions”. In: *Computers in human behavior* 20.2 (2004), pp. 269–288.

BIBLIOGRAPHY

- [127] Hironobu Takagi et al. “Accessibility designer: visualizing usability for the blind”. In: *ACM SIGACCESS accessibility and computing* 77-78 (2003), pp. 177–184.
- [128] Brian Sierkowski. “Achieving web accessibility”. In: *Proceedings of the 30th annual ACM SIGUCCS conference on User services*. 2002, pp. 288–291.
- [129] Wendy Chisholm, Gregg Vanderheiden, and Ian Jacobs. “Web content accessibility guidelines 1.0”. In: *Interactions* 8.4 (2001), pp. 35–54.