

CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Enhancing Web Accessibility Generation with LLM-based IDE Extensions

Master's Thesis in Computer science and engineering

Dantong Luo

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's Thesis 2026

Enhancing Web Accessibility Generation with LLM-based IDE Extensions

Dantong Luo



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Enhancing Web Accessibility Generation with LLM-based IDE Extensions
Dantong Luo

© Dantong Luo, 2026.

Supervisor: Miroslaw Staron, Simin Sun, Department of Computer Science and Engineering
Examiner: Irum Inayat, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Dantong Luo
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

AI coding assistants offer opportunities for web accessibility support, but evidence on how accessibility-focused guidance affects code accessibility and developer experience remains limited. This thesis presents A11yHelper, a Visual Studio Code extension integrated with GitHub Copilot Chat, and compares its accessibility quality, perceived usability, and tool preference with GitHub Copilot and CodeA11y.
The evaluation combined a fixed-prompt experiment and a controlled user study with 18 participants. Both covered five tasks from two repositories. The analysis used task-specific three-level quality ratings, the SUS score, preferences between the two tools used, and descriptive categorization of written reasons.
Quality differences depended on the task. A11yHelper had higher mean scores than GitHub Copilot on form labeling and keyboard navigation in both evaluations, while GitHub Copilot matched or exceeded its means on the remaining tasks. Both scored higher than CodeA11y on the task requiring information distributed across a larger file. Alternative text remained difficult, with only two of 36 user-study artifacts rated Good. Mean SUS scores were 79.38 for GitHub Copilot, 76.46 for A11yHelper, and 59.58 for CodeA11y, preference counts followed the same ordering. The study contributes an integrated accessibility assistant and descriptive evidence of its task-specific strengths and limitations.

Keywords: Web accessibility, AI coding assistants, IDE extensions, usability

Acknowledgements

I express my sincere gratitude to my supervisors, Miroslaw Staron and Simin Sun, for their guidance and insightful feedback throughout this research. I would also like to thank all participants who took part in the empirical evaluation.

Dantong Luo, Gothenburg, October 2026

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

2 Background **3**
2.1 Web Accessibility . . . 3
2.1.1 WCAG . . . 3
2.1.2 Common Accessibility Issues in Web Applications . . . 4
2.2 Automated Accessibility Testing . . . 5
2.3 LLMs in Software Engineering . . . 6
2.3.1 LLMs for Code Generation . . . 6
2.3.2 AI Coding Assistants . . . 6
2.4 Visual Studio Code Extension . . . 6
2.4.1 AI Extensibility in VS Code . . . 7

3 Related Work **9**
3.1 Automated Accessibility Testing in Research . . . 9
3.2 LLM-Based Coding Assistants and Accessibility . . . 10
3.3 Accessibility-Specific AI Development Tools . . . 10
3.4 Workflow Integration . . . 11
3.5 Research Gap . . . 12

4 Methods **13**
4.1 System Design . . . 13
4.1.1 Design Principles . . . 13
4.1.2 Design Decisions . . . 14
4.1.3 System Architecture . . . 15
4.2 System Implementation . . . 16
4.2.1 Implementation Overview and Interaction Model . . . 16
4.2.2 Request Routing and Task Handlers . . . 17
4.2.3 Context Construction . . . 18
4.2.4 Accessibility Scanning . . . 18
4.2.5 Prompt Design . . . 19
4.3 Evaluation Design . . . 20
4.3.1 Tasks and Evaluation Criteria . . . 20

4.3.2 Fixed-Prompt Experiment Design . . . 22
4.3.3 Participants and Group Assignment . . . 22
4.3.4 User Study Procedure . . . 22
4.4 Data Collection and Analysis . . . 23
4.4.1 Background Data . . . 23
4.4.2 RQ1 Analysis: Accessibility Quality . . . 24
4.4.3 RQ2 Analysis: Usability and Preference . . . 24

5 Results **27**
5.1 Participant Background . . . 27
5.2 RQ1: Accessibility Quality . . . 27
5.2.1 Fixed-Prompt Experiment . . . 27
5.2.2 Controlled User Study . . . 29
5.3 RQ2: Usability and Preference . . . 29
5.3.1 SUS Scores . . . 29
5.3.2 Tool Preference . . . 30

6 Discussion **33**
6.1 Interpretation of Findings . . . 33
6.1.1 RQ1: Accessibility Quality . . . 33
6.1.2 RQ2: Usability and Tool Preference . . . 34
6.2 Limitations . . . 35
6.3 Threats to Validity . . . 35
6.3.1 External Validity . . . 35
6.3.2 Internal Validity . . . 36
6.3.3 Construct Validity . . . 37
6.3.4 Conclusion Validity . . . 37
6.4 Future Work . . . 38
6.5 Use of Generative AI Tools . . . 39

7 Conclusion **41**

Bibliography **43**

A Appendix **I**
A.1 Fixed Prompts . . . I
A.2 Consent Form . . . I
A.3 Background Survey . . . II
A.4 System Usability Survey . . . IV
A.5 User Study Guide: AI-Assisted Web Development . . . V
A.6 User-Study Results by Web Development Experience . . . VIII
A.7 User-Study Results by Model Version . . . VIII

List of Figures

2.1 Example of an axe-core rule evaluating accessible naming requirements based on WCAG success criteria [14]. 5

4.1 UML component diagram of A11yHelper. 16
4.2 Interaction workflow of A11yHelper within VS Code. 17

5.1 Accessibility quality scores for the fixed-prompt experiment: (a) mean rubric scores; (b) score distributions. 28
5.2 Accessibility quality scores for the user study: (a) mean rubric scores with bootstrap 95% confidence intervals; (b) distribution of rubric scores. 29
5.3 Distribution of SUS scores for each tool, with $N = 12$ ratings per tool. The dashed line marks 70 as an interpretive reference point [6]. 30
5.4 Overall tool preference ($N = 18$). 31

List of Tables

2.1 Average number of detected accessibility errors per home page in the WebAIM Million analyses, 2022–2026 [50]. 4
2.2 Most common accessibility error types detected in the WebAIM Million 2026 analysis [50]. 4

4.1 Overview of tasks, accessibility issues, and evaluation criteria 21
4.2 BIBD grouping and counterbalanced tool usage order 23

5.1 Participant background ($N = 18$) 28
5.2 Summary statistics of SUS scores by tool ($N = 12$ per tool) 29
5.3 Descriptive categories of preference reasons. 31

A.1 Fixed prompts used for the five evaluation tasks I
A.2 User-study results by web development experience. VIII
A.3 User-study results by model version. VIII

1

Introduction

Web accessibility enables all users, including people with disabilities, to access and use web applications [39]. Accessibility is an essential aspect of software quality and a legal requirement in many countries [57]. The World Wide Web Consortium (W3C) provides the Web Content Accessibility Guidelines (WCAG), which define testable success criteria for accessible web content [2].

Despite the availability of accessibility guidelines, accessibility problems remain widespread in modern web applications. The WebAIM Million 2026 report identified an average of 56.1 accessibility errors per home page across one million home pages [50]. Automated accessibility testing tools such as axe [48] and Lighthouse [16] detect violations through predefined checks based on WCAG [5]. However, these tools provide limited remediation guidance. Developers must interpret the findings and determine appropriate code changes [38].

Large Language Models (LLMs) offer opportunities to support accessibility-related development tasks. LLM-based assistants can analyze code context, understand natural language instructions, and generate code suggestions [41]. They can support remediation by proposing potential fixes and implementation examples [20].

Accessibility issues often occur during the implementation process, motivating provide relevant support directly in the development environment [8]. The Integrated development environments (IDEs) provides an entry point for integrating such support into developers' daily workflows. Through extensibility mechanisms such as plugins and extensions, IDE allows tools to analyze the source code and provide feedback directly during coding [42].

Coding assistants provide one form of support in IDEs. General-purpose coding assistants can provide modification suggestions, but their accessibility quality depends on the prompt and training data [17]. Accessibility-specific assistants such as CodeA11y [37] can provide more targeted support, but limited code context may affect the quality of their suggestions. These limitations motivate the development and evaluation of an assistant that combines accessibility guidance with adaptive code context within the IDE.

This thesis presents A11yHelper, a VS Code extension that runs as a chat participant

in GitHub Copilot Chat. It adds built-in accessibility instructions, an axe scan tool, and file-size-aware context construction. The study evaluates its accessibility quality and perceived usability against GitHub Copilot and CodeA11y.

This research is guided by the following research questions:

- **RQ1:** How does the accessibility quality of web code produced using A11yHelper compare with that produced using GitHub Copilot and CodeA11y across the evaluated tasks?
- **RQ2:** How do developers evaluate A11yHelper, GitHub Copilot, and CodeA11y in terms of perceived usability and overall tool preference?

We compared A11yHelper with GitHub Copilot and CodeA11y in a fixed-prompt experiment and in a user study with 18 participants. Both evaluations covered five web development tasks. The fixed-prompt experiment assessed accessibility quality under predefined prompts, while the controlled user study assessed accessibility quality during developer interaction, perceived usability, and tool preference. Four tasks were adapted from the CodeA11y study [37], and a fifth required relevant code context from different parts of a larger source file.

This thesis makes three contributions:

- The design and implementation of A11yHelper as a VS Code extension integrated into GitHub Copilot Chat.
- A comparative evaluation of A11yHelper, GitHub Copilot, and CodeA11y through a fixed-prompt experiment and a controlled user study across five web development tasks.
- Empirical findings on accessibility quality across tasks, perceived usability, and tool preference for the three assistants.

2

Background

2.1 Web Accessibility

Web accessibility refers to the design and implementation of web applications in a way that enables all users, including people with disabilities, can perceive, understand, navigate, and interact with digital content effectively [39]. When users access the network interface, they may rely on various assistive technologies, such as screen readers, keyboard navigation, voice input systems, or alternative input devices. Ensuring accessibility allows these users to perform essential tasks such as reading content, completing forms, and interacting with interactive components [22].

2.1.1 WCAG

WCAG, developed by W3C, are the most widely adopted technical standard for evaluating and implementing web accessibility [2]. WCAG defines a comprehensive set of testable success criteria that guide developers in creating accessible web content. These criteria are organized under four fundamental principles: Perceivable, Operable, Understandable, and Robust (POUR) [25]. The Perceivable principle requires that users must be able to perceive content through available senses, such as providing alternative text for images and sufficient color contrast. The Operable principle ensures that users can interact with interface components, for example by supporting keyboard navigation and avoiding interaction patterns that require precise pointer control. The Understandable principle requires content and interaction behavior to be predictable and clear, including proper labeling of form elements and consistent navigation structure. The Robust principle ensures compatibility with a wide range of user agents and assistive technologies by promoting semantic HTML and standards-compliant implementation [24].

WCAG has evolved through many versions, including WCAG 2.0, WCAG 2.1, and the more recent WCAG 2.2. Each new version extends the previous version by introducing additional success criteria to address emerging accessibility challenges, such as mobile interaction and cognitive accessibility [52, 2, 54].

WCAG defines three conformity levels: Level A, Level AA, and Level AAA [2]. Level A represents the minimum accessibility requirements, while Level AA contains the recommended standard for most web applications. Level AAA defines the

highest level of accessibility, but may not be feasible for all content types. These guidelines provide a basis for accessibility testing tools and compliance evaluation. Among these, WCAG 2.1 Level AA is cited by several important regulatory frameworks, including the European standard EN 301 549 and recent updates to the Americans with Disabilities Act (ADA) Title II[1, 45].

Many automated accessibility tools use checks based on WCAG success criteria to identify violations [5]. This study uses WCAG 2.1 Level AA as the standard reference for accessibility issue detection and remediation guidance.

2.1.2 Common Accessibility Issues in Web Applications

The WebAIM Million 2026 report identified 56,114,377 accessibility errors across one million home pages, averaging 56.1 errors per page [50]. This represented a 10.1% increase from the 2025 analysis. The proportion of home pages with detected WCAG failures also increased from 94.8% in 2025 to 95.9% in 2026 [50]. Table 2.1 presents the average number of detected errors per home page from 2022 to 2026. These figures describe automatically detected errors rather than the full range of accessibility barriers.

Table 2.1: Average number of detected accessibility errors per home page in the WebAIM Million analyses, 2022–2026 [50].

Year	Average errors per home page
2022	50.8
2023	50.0
2024	56.8
2025	51.0
2026	56.1

Table 2.2 presents the six most common accessibility error types detected in the WebAIM Million 2026 analysis and the percentage of home pages containing each type [50]. Low contrast text was the most common issue, detected on 83.9% of home pages. Together, these six categories accounted for 96% of all detected errors [50]. These issues can hinder users who rely on assistive technologies from accessing and interacting with web content [56].

Table 2.2: Most common accessibility error types detected in the WebAIM Million 2026 analysis [50].

Error type	Percentage of home pages
Low contrast text	83.9%
Missing alternative text for images	53.1%
Missing form input labels	51.0%
Empty links	46.3%
Empty buttons	30.6%
Missing document language	13.5%

2.2 Automated Accessibility Testing

Most automated accessibility testing tools use predefined rules derived from WCAG success criteria [4]. Each rule specifies an accessibility requirement and a method for checking whether web content meets that requirement.

Tools such as axe and Lighthouse follow this rule-based approach [48, 16]. The axe accessibility engine, developed by Deque Systems, checks web content against accessibility rules based on WCAG success criteria and produces structured reports describing detected violations. Axe is commonly integrated into browser developer tools, automated testing pipelines, and development environments [26]. Google Lighthouse also performs accessibility audits on rendered web pages using checks based on the axe engine [13].

During analysis, the testing engine parses the web page structure and applies predefined rules to relevant elements. Figure 2.1 shows an example of a rule implementation in the axe-core accessibility engine that checks whether a web element has a valid accessible naming method, such as alternative text or an associated label. The rule analyzes the element's properties and DOM structure, and reports an accessibility violation if the naming requirement defined by WCAG is not satisfied.

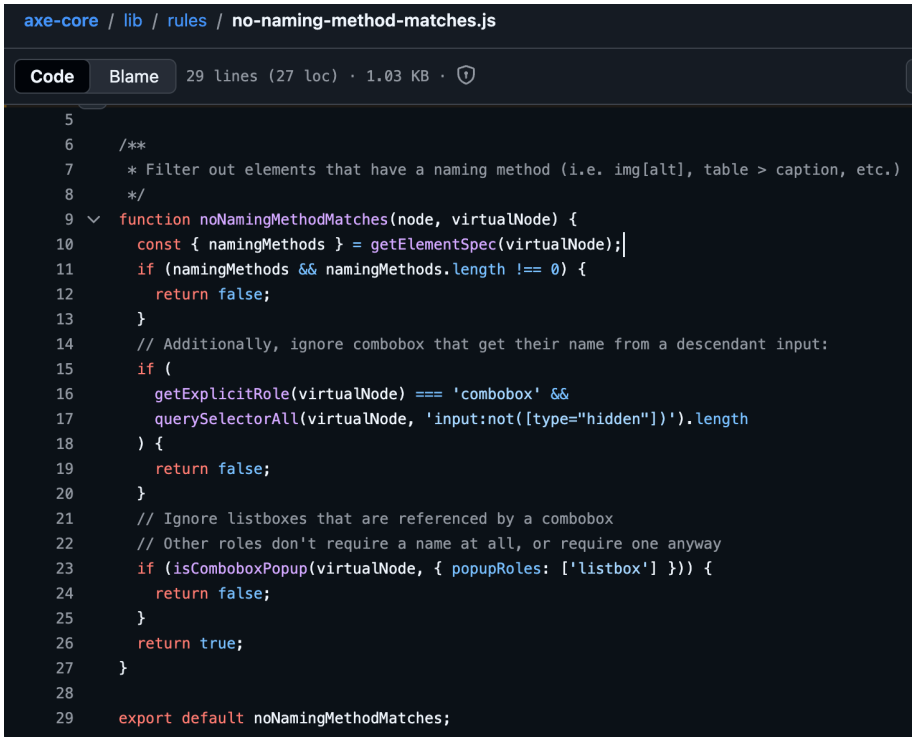


Figure 2.1: Example of an axe-core rule evaluating accessible naming requirements based on WCAG success criteria [14].

Because rule-based analysis is deterministic, these tools provide consistent and reproducible results. Integration into automated testing pipelines allows developers to

repeat accessibility checks during software validation [11]. These checks complement manual inspection by identifying violations covered by the configured rules.

Rule-based analysis has limitations when accessibility requirements involve semantic or contextual interpretation [37]. For example, checking whether an image has an alternative text attribute does not establish whether the text accurately describes the image or serves its purpose in the surrounding content. Evaluating these aspects requires information beyond the presence of the attribute.

2.3 LLMs in Software Engineering

2.3.1 LLMs for Code Generation

LLMs are deep neural network models trained on large-scale corpora that can include natural language and source code. Transformer-based models use self-attention mechanisms to represent relationships between tokens in an input sequence [47]. During pretraining, autoregressive models learn to predict the next token based on preceding context [41]. Instruction tuning further trains models to respond to explicit instructions [58]. In software engineering, LLMs can generate code snippets and modification suggestions from programming-related inputs [41].

For code generation and modification, the model input can include task instructions and source code context. The instructions specify the requested operation, while the code context provides information about the existing implementation [41]. LLM-based approaches can use this input to generate candidate code changes, whereas rule-based detection applies predefined checks to identify violations.

2.3.2 AI Coding Assistants

AI coding assistants integrate language models into development environments to support code completion, explanation, and modification [46]. GitHub Copilot is a general-purpose coding assistant available in IDEs including VS Code and provides inline code suggestions based on surrounding code context [18]. GitHub Copilot also provides a chat interface through which developers can ask questions and request explanations or code modifications [19].

2.4 Visual Studio Code Extension

An IDE combines development tools within a unified interface, including source code editing, debugging, and version control integration [42]. Traditional IDEs such as Eclipse, IntelliJ IDEA, and Visual Studio offer extensive built-in functionality tailored for specific programming languages and frameworks. These environments often provide deep integration with compilers, debuggers, and language-specific analysis tools.

Visual Studio Code (VS Code) is a lightweight cross-platform source code editor

developed by Microsoft. It provides core development features and supports additional programming languages, frameworks and functionality through a modular extension system [35]. This design enables developers to customize the development environment according to the needs of the project. VS Code supports Windows, macOS and Linux platforms.

VS Code extensions run in an extension host that communicates with the main application [34]. Extensions interact with VS Code through the Extension API, which provides access to editor state, open documents, and workspace information. The API also allows extensions to register commands and create user interface components such as panels and views [33].

2.4.1 AI Extensibility in VS Code

VS Code provides APIs that allow extensions to integrate language models and domain-specific capabilities into the development environment [32]. The Chat API enables extensions to implement chat participants that handle user requests and responses within the chat interface. Users can invoke a participant through an `@`-mention. The Language Model API provides programmatic access to language models, supporting model requests from chat participants and other extension features.

Language model tools provide specialized operations, such as retrieving information or invoking external services. Extensions implement these operations through the Language Model Tools API. These tools can be used in agent mode or invoked through tool-calling workflows implemented by chat participants.

3

Related Work

3.1 Automated Accessibility Testing in Research

Previous studies evaluated the effectiveness and coverage of automated accessibility testing tools. Abduganiev compared eight tools for evaluating web accessibility against WCAG 2.0. The results showed that individual coverage rates ranging from 12.7% to 32.4%, averaging approximately 20% [4]. This indicates that although automated testing can provide scalable support, its coverage remains constrained, and no individual tool can detected all accessibility issues.

Despite their effectiveness, automated accessibility testing tools have limitations in semantic understanding. Rule-based checks can verify whether accessibility-related attributes exist, but cannot fully address accessibility requirements that depend on content meaning and context [37]. For example, checking whether an image has an alternative text attribute does not establish whether the description conveys the relevant visual information [5].

Another limitation is the readability of the generated analysis results. Detection results also require interpretation before developers can apply code changes. The reported issues typically contain technical descriptions without clear prioritization or actionable remediation steps, requiring developers to manually interpret the results and map them to appropriate fixes [49]. This process introduces additional cognitive load and increases the effort required to address accessibility issues, particularly for developers without specialized accessibility knowledge [38].

From a development perspective, these limitations mean that developers are still responsible for transforming accessibility test results into specific code modifications. Previous studies have shown that this manual interpretation process may increase the cognitive burden and interrupt the development workflow, especially for developers with limited knowledge of accessibility [8].

These challenges have motivated recent studies to explore the use of LLMs to provide accessibility support. The LLM-based method can supplement the traditional automated accessibility test by identifying issues under WCAG success criteria, which usually requires manual evaluation [28].

3.2 LLM-Based Coding Assistants and Accessibility

General-purpose LLM-based coding assistants can provide accessibility-related improvements, but their suggestions depend heavily on training data and surrounding code context [18]. These systems learn from open-source repositories with varying code quality, so their outputs may reflect inconsistent accessibility practices. For example, when enhancing the contrast between a button and its background, Copilot can identify low-contrast color combinations and suggest alternative values. However, the suggested colors may not strictly satisfy WCAG-recommended contrast ratios [36]. The accessibility quality of generated code also depends on the surrounding code. If that code lacks semantic structure or accessibility attributes, the model may reproduce these patterns.

Previous studies pointed out that there are some limitations in how developers prompted Copilot and used its suggestions. Developers must manually identify accessibility issues and request assistance, which requires them to have a certain knowledge and awareness of accessibility [44]. A previous study found that participants mainly focused on visual and functional modifications in prompts, without explicitly requiring accessibility features [37]. When the prompt does not contain accessibility requirements, the generated suggestions may not fully meet these requirements. In addition, developers may trust the results generated by AI without checking the correctness. This reliance on developer prompts limits consistent accessibility support from general-purpose coding assistants during routine development.

These limitations motivate the development of accessibility-specific AI assistants that aim to provide consistent guidance.

3.3 Accessibility-Specific AI Development Tools

Previous work explored LLM-based detection of accessibility issues that typically require manual assessment. López-Gil and Pereira developed scripts to evaluate three WCAG success criteria: Non-text Content, Link Purpose (In Context), and Language of Parts. Using selected and modified test cases based on WCAG Accessibility Conformance Testing rules, they reported an overall detection rate of 87.18%. The scripts identified issues that the automated evaluators in the study missed or reported only as warnings [28].

For web accessibility detection, He et al. developed GenA11y, which extracts relevant page elements and uses LLM prompts to evaluate them against WCAG success criteria. The authors reported a precision of 94.5% on twelve real-world websites [21].

Previous work also investigated accessibility repair in mobile apps. Zhang et al. developed Iris to repair color-related accessibility issues in Android apps through automated and context-aware repair method [59]. Mehralian et al. developed FixAlly, a multi-agent LLM system that localizes scanner-reported issues and generates

code modification suggestions for iOS apps. FixAlly achieved 77% effectiveness in generating plausible fix suggestions for issues identified by accessibility scanners [31].

CodeA11y represents the most directly related work to this study. Like A11yHelper, it provides accessibility-specific assistance as a chat participant in GitHub Copilot Chat within VS Code. Its architecture comprises three specialized agents. The Responder Agent generates accessibility-aware code suggestions, the Correction Agent provides additional suggestions based on axe DevTools Accessibility Linter logs, and the Reminder Agent prompts developers to complete manual steps, such as replacing placeholder attributes. The agents use cursor-centered code, chat history, accessibility linter logs, and project information from README and index files [37].

The CodeA11y authors reported that the Copilot plugin infrastructure limited the source code available to their implementation. Its code context comprised 100 lines centered around the cursor, and the implementation filtered the collected context when it exceeded 4,000 characters [37]. A11yHelper instead selects code context according to the active file's size, retaining full content for smaller files and combining header and cursor-local content for larger files.

Mowar et al. evaluated CodeA11y against GitHub Copilot in a within-subjects study with 20 novice developers. Participants using CodeA11y achieved higher accessibility scores for form labels, button contrast, and alternative text, while the study found no statistically significant improvement in link labeling [37]. The present study adapted its four tasks and added a fifth task involving a larger source file. It also compared A11yHelper, GitHub Copilot, and CodeA11y through both a fixed-prompt experiment and a controlled user study.

3.4 Workflow Integration

Previous work examined how interruptions affect software development. Tools such as browser extensions, standalone scanners, and CLI-based checkers require developers to leave the code editor, run a separate analysis, and then return to the editor to apply fixes. Research shows that interrupted developers tend to compensate by working faster, but at the expense of increased stress and reduced concentration [30]. Abad et al. found that task switching disrupted development tasks, with effects varying according to the interruption context [3]. Ritonummi et al. identified the absence of distractions and interruptions as one factor supporting developers' flow experience [43]. These studies provided a basis for considering interaction demands when evaluating development tools.

CodeA11y's evaluation illustrates challenges in presenting feedback within the IDE. The authors observed that some participants missed floating reminders because VS Code dismissed them more quickly than expected. After eight participants, they changed the reminders to modal dialogs. Participants initially attended to these dialogs but later began closing them without completing the suggested steps. The authors nevertheless reported that reminders were somewhat effective and cautioned

that the small sample limited comparisons between the two notification strategies [37].

These observations motivate examining developer experience alongside the accessibility quality of generated code. The present study measured perceived usability using SUS and collected overall tool preferences with written explanations.

3.5 Research Gap

Previous software development accessibility support methods provide different forms of assistance, but the above discussion also points out that these methods have limitations in the following aspects.

- **From Detection to Remediation:** Automated accessibility testing tools are effective in detecting rule-based violations, but they provide limited explanations and lack practical repair guidance. Developers must interpret the test results by themselves and convert them into corresponding code modifications.
- **Accessibility Support in General-Purpose Assistants:** General-purpose LLM-based coding assistants enable contextual code generation within IDEs, but their accessibility support remains unreliable. Their outputs depend on training data and surrounding code context, and accessibility improvements are typically only introduced when explicitly requested by users.
- **Context Window Constraints:** CodeA11y combines accessibility-specific guidance with linter feedback. Its authors nevertheless reported restrictions on the source code available to their implementation and filtering of collected context [37]. This leaves room to investigate alternative context selection strategies, without assuming that a larger input necessarily produces better suggestions.
- **Workflow Disruption:** In addition to the above limitations, workflow integration is still a common challenge facing these methods. Traditional accessibility tools usually run independently of the development environment, requiring developers to switch between code editors and analysis reports. This separation can increase the effort required to address accessibility issues.

These limitations informed the development of A11yHelper, a VS Code extension integrated into GitHub Copilot Chat. The system combines automated accessibility detection with LLM-based guidance and code modification suggestions.

4

Methods

This chapter presents the methodology of the study, including the design and implementation of A11yHelper and the evaluation approach used to assess its accessibility quality, perceived usability, and tool preference across the three assistants. The replication package is available at `https://github.com/A11yHelper/Replication-Package`.

4.1 System Design

This section describes the design of A11yHelper and explains how the limitations identified in the previous chapters are translated into system-level solutions.

4.1.1 Design Principles

The limitations discussed in the background and related work, together with engineering judgments about how to organize the assistant's functionality, informed five design principles that guided the development of A11yHelper. The principles address five aspects of accessibility-oriented development support: accessibility-specific guidance, integration with the developer workflow, task-oriented interaction, interpretable accessibility reporting, and controlled context construction.

P1: Accessibility-specific Guidance. General-purpose LLM coding assistants may omit accessibility requirements unless developers explicitly request them. P1 requires the system to provide accessibility-specific guidance without relying on developers to formulate these requirements themselves. The guide should be based on the established accessibility standards. When analyzing or generating code, the assistant should prioritize accessibility.

P2: Integration with the Developer Workflow. P2 requires accessibility support to be available directly within the developer's existing coding workflow. Developers should be able to request accessibility analysis and remediation guidance without leaving the development environment or adopting a separate interaction workflow. The principle aims to enable developers to get accessibility feedback when making code implementation decisions.

P3: Task-oriented Interaction Design. Accessibility support involves distinct

developer tasks, including issue detection, code remediation, and requests for accessibility guidance. These tasks require different information and model assistance methods. P3 requires the system to divide common accessibility interactions into independent task categories with clear responsibilities. The system should provide corresponding information and instructions according to the purpose of each task, and return the response related to the requested operation.

P4: Structured and Interpretable Accessibility Reporting. Developers need to link the detected accessibility issues with relevant requirements and specific code modifications. P4 requires the system to present the detected accessibility issues in an understandable and actionable form, rather than just showing the original scan output. The system should help developers determine which issues to prioritize, understand why these issues are important, and identify appropriate repair steps. The information presentation should retain relevant technical information, and omit details that are not directly related to the repair work.

P5: Controlled Context Construction. LLM-based coding assistants can process only a limited amount of input in each interaction. Including an entire large file may consume substantial context capacity, while providing only a narrow code fragment may omit structural information needed to understand the current implementation. P5 requires the system to control the amount of source code included in the model input while retaining local and file-level information relevant to the current request.

4.1.2 Design Decisions

Five design decisions were adopted to address the design principles. These decisions cover accessibility-oriented prompts, GitHub Copilot Chat integration, task-specific handlers, prioritized scan summaries, and file-size-aware context construction. They connect the design requirements to the system architecture described in Section 4.1.3.

D1: Accessibility-oriented Prompt Engineering. To address P1, accessibility-oriented prompt engineering was adopted to make accessibility requirements explicit in model interactions. The shared accessibility prompt assigns the model the role of an accessibility coach and instructs it to prioritize accessible implementation when analyzing or generating code. Accessibility constraints and references to WCAG 2.1 Level AA guide the model toward established requirements rather than relying only on the user's prompt or surrounding code. This decision ensures that the same accessibility-oriented instructions are provided across model interactions.

D2: Implementation as a GitHub Copilot Chat Participant. To address P2, A11yHelper was implemented as a domain-specific participant within GitHub Copilot Chat. This approach allows developers to invoke accessibility support through the same chat interface used for AI-assisted development rather than through a separate application or panel.

D3: Routing-based Architecture with Task-specific Handlers. To address P3, a routing-based architecture was adopted with dedicated handlers for three task categories. A shared entry point receives user requests and assigns each request to the handler responsible for the corresponding task. Each handler defines the workflow and prompt configuration for its task, allowing scanning, remediation, and general accessibility guidance to remain logically separated. This structure was intended to keep task-specific processing separate and provide a basis for adding further task categories.

D4: Prioritized Accessibility Analysis with Actionable Guidance. To address P4, a simplified scan summary was adopted as the model input instead of the complete scanner output. The scanning tool is still responsible for the detection work, while A11yHelper constructs an overview of the detected violations and a prioritized issue list. Violations are ranked using their impact level and affected node count, and only the highest-ranked issues are retained in the list. Each retained issue includes a short explanation of why it matters and a hint on how to fix it.

D5: File-size-aware Context Construction To address P5, file-size-aware context construction was adopted to control the source code included in the model input. A11yHelper adapts this representation to the size of the active file. For a small file, it includes the full content subject to a character limit. For a larger file, it combines a header block from the beginning of the file with a code window around the cursor. This strategy combines cursor-local code with file-level information while limiting the amount of source code supplied. The model input also includes basic dependency information from the workspace.

4.1.3 System Architecture

Figure 4.1 presents the component-level architecture of A11yHelper and the main flow of a developer request through the system. The Chat Participant receives requests from GitHub Copilot Chat and passes them to the Router, which directs each request to one of three task-specific handlers: Scan, Fix, or Help. Each handler addresses a distinct form of accessibility support while drawing on a common set of shared resources.

The Context Builder supplies the source code and project dependency information used to construct the model input. The Prompt Library supplies shared accessibility instructions and task-specific prompts, while Settings provides extension configuration. For scan requests, the Scan Handler additionally calls Scan Support, which invokes axe CLI and converts the detected violations into a prioritized summary.

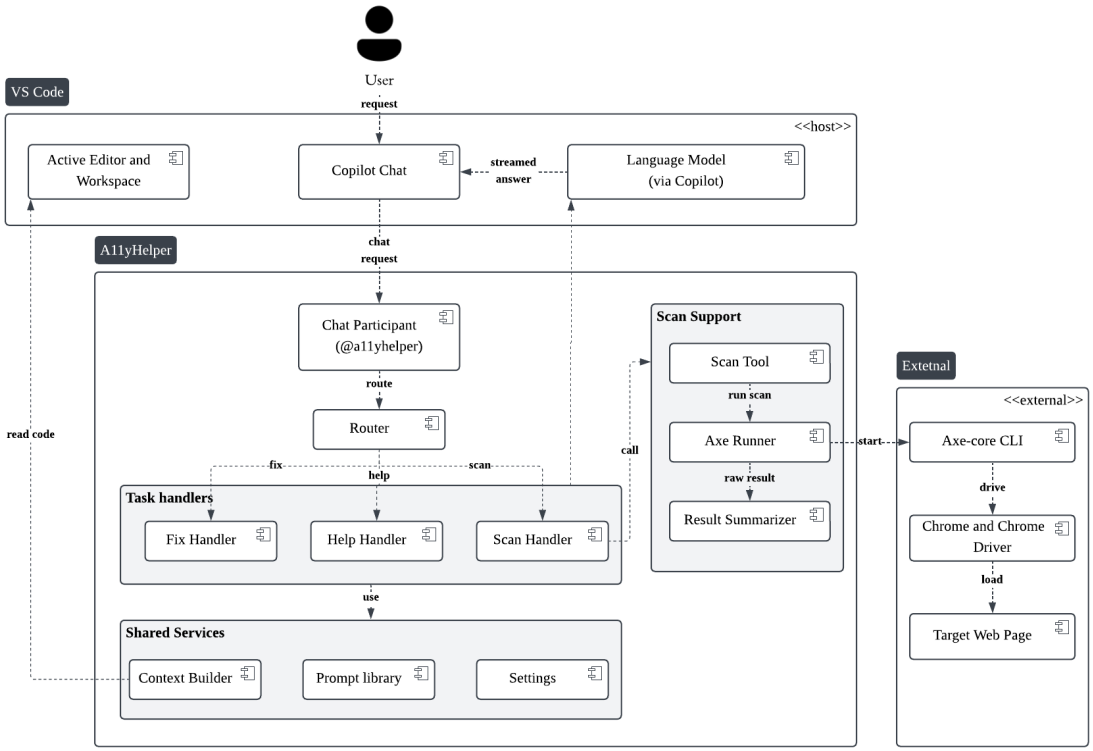


Figure 4.1: UML component diagram of A11yHelper.

4.2 System Implementation

This section presents how these design decisions are implemented in the VS Code extension.

4.2.1 Implementation Overview and Interaction Model

Figure 4.2 illustrates how developers interact with A11yHelper within the VS Code environment. The extension is registered as a GitHub Copilot Chat Participant and is invoked through `@a11yhelper`. Responses are streamed to the GitHub Copilot Chat interface, where developers can review the generated output and use the platform's native editing functions to apply suggested code changes to the active file.

(1) Active Editor: The active editor provides the source code context used by A11yHelper during accessibility-related tasks. The assistant reads the current file and relevant surrounding code through VS Code API.

(2) Inline Annotations for Guided Fixes: When generated code requires developer input or verification, A11yHelper includes inline `a11y-helper TODO` or `a11y-helper FIXME` annotations. TODO annotations indicate cases that require additional product or content information, while FIXME annotations mark suggestions that should

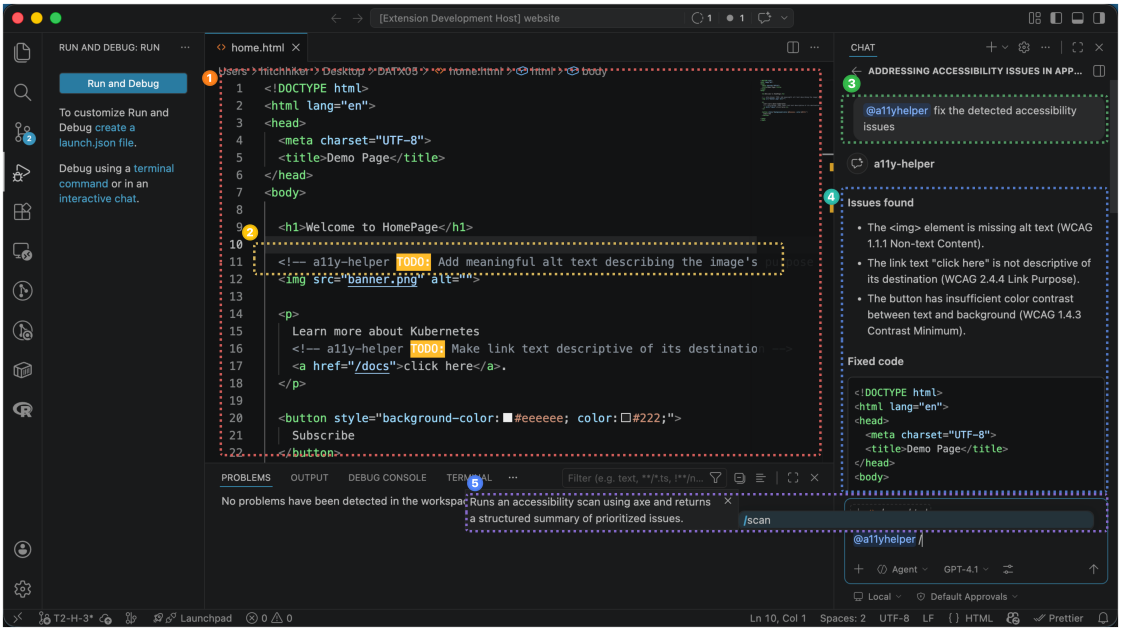


Figure 4.2: Interaction workflow of A11yHelper within VS Code.

be verified by the developer.

(3) Assistant Invocation via GitHub Copilot Chat (`@a11yhelper`): Developers invoke A11yHelper through the `@a11yhelper` mention in GitHub Copilot Chat.

(4) Accessibility Analysis and Remediation Output: Depending on the request, the response may identify accessibility issues, explain relevant accessibility requirements, and provide remediation guidance or example code.

(5) Accessibility Scanning via `/scan`: Developers use the `/scan` command to request accessibility scanning of a target page. Detected violations are processed into a prioritized summary that the model can use to explain accessibility issues and suggest remediation.

4.2.2 Request Routing and Task Handlers

A11yHelper implements request routing through lightweight rule-based matching. The router first checks explicit commands. A `scan` command is assigned to the Scan Handler, while a `fix` command is assigned to the Fix Handler. When no matching command is present, the router examines the lowercased request text. Requests beginning with `scan` followed by a space or containing expressions such as `run axe`, `axe scan`, or `run /scan` are classified as scan requests. Requests containing the substring `fix` are classified as fix requests. All remaining requests are handled by the Help Handler.

All three handlers collect code context and combine shared accessibility instructions with a task-specific prompt. The Scan Handler obtains a target URL from con-

figuration and asks the developer to review or change it when URL confirmation is enabled. The Fix Handler uses instructions for localized remediation, while the Help Handler covers explanations and general code assistance. All handlers pass the user request and the prepared context and instructions to the same chat request function.

4.2.3 Context Construction

Before invoking the language model, each handler collects a context snapshot from the active editor and the current workspace. Language models do not always use information in long inputs reliably [27]. A11yHelper adapts the source code representation to the size of the active file. For files of at most 300 lines, it includes the full source code up to a limit of 24,000 characters. For larger files, it builds a reduced representation from two regions. The first is a header block drawn from the first 60 lines, retaining imports, exports, comments, and adjacent structural lines. The second is a code window covering up to 80 lines before and 80 lines after the cursor. This representation retains code near the cursor together with selected file-level information.

These thresholds are default parameters according to experience. The purpose is to balance the amount of source code provided to the model with the need to limit input size. The 300-line threshold was chosen together with the limit of 24,000 characters, using a conventional line width of approximately 80 characters as a reference. The upper limit of the character can also prevent individual long lines of code from causing excessive input.

The context also includes project dependency information obtained from the workspace `package.json`. A11yHelper combines `dependencies` and `devDependencies`, with entries from `devDependencies` taking precedence when the same package occurs in both sections.

4.2.4 Accessibility Scanning

A11yHelper implements a scan tool that connects the handler to the existing axe CLI scanner. For a scan request, A11yHelper runs axe CLI against the target page, processes the detected issues, and then returns a structured summary to the model. The model then uses this summary to explain the issues and suggest possible fixes.

A11yHelper converts raw axe violations into a `ScanSummary` containing an overview and a prioritized issue list. Each violation rule receives a priority score:

$$p_i = 100w_i + n_i,$$

where n_i is the number of affected nodes reported for rule i. A11yHelper maps axe-core's impact categories, which describe the severity of accessibility issues [15], to weights w_i of 4 for critical, 3 for serious, 2 for moderate, and 1 for minor.

The score combines issue severitywith the number of affected nodes. The coefficient of 100 assigns a gap of 100 points between adjacent influence level, and each affected

node adds one point. Therefore, when the difference in the number of nodes is small, it affects the ranking, but the difference in the number of nodes is large enough to reverse the order. For example, when a rule with a lower impact level affects more than 100 nodes than a neighboring rule with a higher level, it ranks higher.

A11yHelper sorts rules by descending score, breaking ties by descending node count, and retains the first Nrules. The configurable N default value is 3, which is used to provide a brief list of problems for preliminary inspection. Each retained issue includes its rule identifier, impact, affected node count, remediation guidance, and up to two representative nodes. The purpose of these restrictions is to keep the detailed summary concise and readable.

4.2.5 Prompt Design

Each handler stitches the shared accessibility prompt, its task-specific prompt, and the formatted context. It passes this text and the original user request to `chatUtils.sendChatParticipantRequest`, using the model supplied by the current chat request. The scan handler also provides the scanning tool. The `chat-extension-utils` library orchestrates the request and streams the response to GitHub Copilot Chat.

The shared prompt assigns an accessibility coach role and treats accessibility as a default requirement for code generation and modification. It prioritizes semantic HTML, accessible names, keyboard operability, and appropriate ARIA use. The role assignment and prescribed output formats correspond to the Persona and Template patterns described by White et al. [51]. Listing 4.1 shows selected instructions. The prompt also lists WCAG 2.1, WAI-ARIA Authoring Practices, and MDN as reference sources.

```
1  You are an accessibility coach and engineering assistant.
2  Your goal is to help developers write and improve web code that
       aligns with WCAG 2.1 Level AA.
3
4  Rules:
5  - Treat accessibility as a default implementation requirement,
       not an optional follow-up.
6  [...]
7  - Prefer semantic HTML, correct form labels, keyboard
       operability, clear accessible names, and appropriate heading
       structure.
8  - Use ARIA only when needed. Follow the WAI-ARIA Authoring
       Practices.
9  [...]
10 - Preserve the developer intent and existing structure when
       practical.
11 [...]
12 - Do not invent WCAG success criterion numbers or unsupported
       details.
```

```
13 [...]
```

Listing 4.1: Shared accessibility prompt.

4.3 Evaluation Design

The evaluation compared A11yHelper with GitHub Copilot and CodeA11y through a fixed-prompt experiment and a controlled user study. The fixed-prompt experiment compared accessibility quality using the same task prompts across the three tools, controlling variation in request wording. The user study assessed accessibility quality when participants freely interacted with their assigned tools, as well as perceived usability and overall tool preference.

GitHub Copilot and CodeA11y were selected as baselines, because both of them can be used within VS Code and support code generation or modification in response to natural language requests. GitHub Copilot serves as a general-purpose baseline, while CodeA11y provides an accessibility-focused comparison in a similar interactive environment.

4.3.1 Tasks and Evaluation Criteria

Both evaluations used the same five web development tasks, covering different accessibility requirements, as summarized in Table 4.1. The first four tasks were taken from the CodeA11y study without modification [37] and used the same two open-source repositories, Kubernetes [23] and BBC News [7]. These repositories differ in structure and technology stack, providing two development contexts for the evaluation.

T5 was introduced to assess tool performance on a task requiring information from different parts of a larger source file. T5 comes from an issue in the Kubernetes repository [1]. Following the task selection approach of the CodeA11y, T5 was based on real project issues involving accessibility requirements, and evaluates with task-specific rubric [37]. This task requires adding a Bluesky community card within an existing `div#bluesky` element, including a clickable logo image and linked title text that follow the structure and formatting of sibling community cards. Its accessibility evaluation involves consistent identification, which requires components with the same functionality to be identified consistently under WCAG 2.1 [53]. Unlike T1 to T4, the relevant information required by T5 is scattered in different locations of the file, rather than concentrated in a small code area.

The assessment used the three quality levels of Unacceptable, Average, and Good adopted by CodeA11y from Pillai et al. [37, 40]. Each level has task-specific criteria, as shown in Table 4.1. The criteria for T1 through T4 were taken from CodeA11y without modification. For T5, the same three-level structure was applied to the consistency of the added community card with its sibling cards.

[1] `https://github.com/kubernetes/website/issues/49547`

Table 4.1: Overview of tasks, accessibility issues, and evaluation criteria

Task	Difficulty	Issue	Evaluation Criteria
T1: Button Visibility	Easy	Color Contrast	**Unacceptable**: contrast ratio $<$ 4.5:1 (normal text) or $<$ 3:1 (large text). **Average**: WCAG AA contrast in default state ($\geq$ 4.5:1). **Good**: WCAG AA contrast in all states (default, hover, active, focus).
T2: Form Element	Moderate	Form Labeling	**Unacceptable**: missing form labels and keyboard navigation. **Average**: either form labels or keyboard navigation implemented. **Good**: both form labeling and keyboard navigation implemented.
T3: Add Section	Moderate	Link Labeling	**Unacceptable**: missing link descriptions. **Average**: uninformative link descriptions. **Good**: descriptive link descriptions.
T4: Enhance Image for SEO	Difficult	Alt-text	**Unacceptable**: missing or uninformative alt-text. **Average**: alt-text with $<$ 3 required descriptors [29]. **Good**: alt-text with $\geq$ 3 out of 4 required descriptors.
T5: Add Bluesky Community Card	Moderate	Consistent Identification	**Unacceptable**: missing card or inconsistent identification. **Average**: card implemented with partial inconsistency. **Good**: correctly implemented card consistent with sibling community cards[53].

4.3.2 Fixed-Prompt Experiment Design

The fixed-prompt experiment compared the three tools using the same prompt for each of the five tasks. Each prompt described the task requirement and was provided to GitHub Copilot, CodeA11y, and A11yHelper. The complete prompts are provided in Appendix A.1.

GitHub Copilot, CodeA11y, and A11yHelper all used GPT-4.1 through GitHub Copilot Chat in Ask mode. The selected model, interaction mode, and prompts for each task were consistent with different tools and runs. The experiment used CodeA11y version 0.0.5 from the VS Code Marketplace. Each tool and task combination is run three times separately. Before each run, the source code was restored to the initial state, and the chat session was also cleared.

The resulting code was assessed using the task-specific criteria described in Section 4.3.1. The fixed-prompt experiment provided a comparison under the predefined prompt and supplemented the user research results.

4.3.3 Participants and Group Assignment

A11yHelper targets developers who lack specialised accessibility knowledge. We therefore recruited developers with web development experience who use AI coding assistants in their regular development workflows. The inclusion criterion required prior experience with HTML, CSS, and JavaScript, but did not require formal accessibility training. Eighteen participants took part in the study and were recruited primarily from university populations through emails and social media platforms.

The user study used a balanced incomplete block design (BIBD) [55, 9]. The three tools were the treatments, and each participant used two of them, forming an incomplete block. The design included all three possible tool pairs. Participants were randomly assigned to the six subgroups shown in Table 4.2, with three participants in each subgroup. Each tool pair therefore included six participants, with three using each of the two tool orders. This counterbalancing distributed tool order evenly across participants.

Each participant completed the same five tasks with both assigned tools, keeping the task requirements consistent across tool sessions. The grouping provided 12 participant evaluations per tool and six paired comparisons for each tool pair.

4.3.4 User Study Procedure

Before the experiment, each participant signed an informed consent form and completed the background questionnaire, then watched a short introductory video and received written instructions. The complete background questionnaire is provided in Appendix A.3. During the introduction for participants, the study was described as an evaluation of two general coding assistants. The introduction explained how to use each assigned tool and how to apply the generated suggestions in the editor.

Table 4.2: BIBD grouping and counterbalanced tool usage order

Group	Subgroup	Tool Usage Order	Participants
G1	G1a	A11yHelper → CodeA11y	3
	G1b	CodeA11y → A11yHelper	3
G2	G2a	A11yHelper → GitHub Copilot	3
	G2b	GitHub Copilot → A11yHelper	3
G3	G3a	GitHub Copilot → CodeA11y	3
	G3b	CodeA11y → GitHub Copilot	3

The research introduction did not provide guidance related to accessibility or tips for specific tasks.

Participants completed the five development tasks with each assigned tool, following the counterbalanced tool order determined by their subgroup. They could complete the tasks in any order within each tool session and freely interact with the assigned tool, including submitting multiple requests and follow-up questions. Each tool session was intended to last approximately 30 minutes. The researcher observed the session and provided assistance only for technical issues, such as environment setup problems, without offering guidance on task solutions or accessibility implementation.

Immediately after completing the tasks with each tool, participants completed the System Usability Scale (SUS), a standardized ten-item questionnaire for perceived usability [10]. The complete SUS questionnaire is provided in Appendix A.4. After using both assigned tools, participants selected which of the two they preferred overall and provided a written explanation for their choice.

4.4 Data Collection and Analysis

This section describes the data collected and the analysis used to answer the research questions. Background information characterized the sample, code artifacts supported the assessment of accessibility quality for RQ1, and SUS responses, tool preference choices, and written reasons addressed RQ2.

The study mainly focused on the final code artifacts and questionnaire responses. Time on task was not recorded, because participants repeated the same five tasks with two tools, and the learning or order effect were expected to make the comparison between different tools more complicated. To limit the interference to the participants' natural workflow and protect their privacy, this study did not record the interactions count, suggestion acceptance and chat logs.

4.4.1 Background Data

The background questionnaire collected participants' current status, web development experience, self-reported familiarity with HTML, CSS, and JavaScript, fre-

quency of AI coding assistant use, and familiarity with accessibility concepts such as WCAG or ARIA. Participants reported web development experience in year ranges and rated familiarity with each web technology on a five-point scale from 1 (Not familiar) to 5 (Very familiar). AI coding assistant use and accessibility familiarity used five and four response categories, respectively. Appendix A.3 provides the complete questionnaire.

The analysis summarized current status, experience ranges, AI coding assistant use, and accessibility familiarity using counts and percentages, with 18 participants as the denominator. Self-reported familiarity ratings for HTML, CSS, and JavaScript were summarized using means and standard deviations. These summaries described the study sample and provided background for interpreting the RQ1 and RQ2 results in next chapter.

4.4.2 RQ1 Analysis: Accessibility Quality

To answer RQ1, the assessment used the code after applying the generated changes in the fixed-prompt experiment and the final file saved for each task in the controlled user study. Each user-study artifact corresponded to one participant, one task, and one tool condition. All expected task artifacts received a score. Incomplete solutions were retained and assessed against the same task-specific criteria.

Each artifact was evaluated according to the criteria in Table 4.1. For T1–T3 and T5, a JavaScript scoring script was developed to identify relevant elements and calculate the score according to the task-specific rubric and axe CLI results. The researcher and the thesis supervisor then reviewed the scoring results against the rubric. For T4 which required assessing the quality of alternative text, the researcher and the thesis supervisor jointly evaluated it using the predefined criteria. Each output received one of three levels: Unacceptable, Average, or Good, mapped to numeric scores of 0, 1, and 2, respectively.

Before scoring, the researcher assigned each artifact an anonymized record number and concealed participant identity and tool condition in both evaluations. During scoring, the evaluator did not know the participant's identity or the tool associated with the artifact.

The analysis summarized scores separately for the two evaluations by task and tool condition. For the fixed-prompt experiment, each mean combined three scores from the corresponding tool-task pair. For the controlled user study, each mean combined the scores of the 12 participants who completed that task with the corresponding tool. As a supplement, the analysis also reports the counts and proportions of scores of 0, 1, and 2, as well as the bootstrap 95% confidence interval for the mean.

4.4.3 RQ2 Analysis: Usability and Preference

To answer RQ2, the analysis used SUS responses, overall tool preference choices, and written preference reasons. Participants rated the ten SUS items on a five-point scale

from 1 (Strongly disagree) to 5 (Strongly agree) [10]. Following the standard scoring procedure, the researcher subtracted one from each odd-numbered item response and subtracted each even-numbered item response from five. These item numbers refer to the ten SUS statements, excluding the tool identification question. The adjusted values were summed and multiplied by 2.5, producing a score from 0 to 100 for each participant's evaluation of one tool. The analysis summarized the 12 evaluation scores of each tool by calculating the average, median and standard deviation.

After using both assigned tools, each participant selected the tool they preferred overall and provided a written reason. Participants chose only between the two tools they had used. The analysis summarized preference using counts and percentages across the 18 participants. Each tool was available for selection by 12 participants under the grouping described in Section 4.3.3.

The researcher reviewed all 18 written responses and grouped similar reasons into descriptive categories derived from the responses. Each response was assigned to one category. This descriptive categorization summarized participants' reported reasons for their preferences.

5

Results

5.1 Participant Background

Eighteen participants took part in the controlled user study. As shown in Table 5.1, most were master's students (13, 72.2%). The remaining participants included two PhD students, one bachelor's student, one industry professional, and one backend developer.

Most participants reported two years or less of web development experience. Eight participants (44.4%) reported less than one year of experience, and five (27.8%) reported one to two years, together accounting for 72.2% of the sample. Only one participant (5.6%) reported more than five years of experience. The mean self-reported familiarity ratings for HTML, CSS, and JavaScript were 2.89, 2.72, and 2.67, respectively, on a five-point scale. The corresponding standard deviations were 1.28, 1.49, and 1.50.

All participants reported using AI coding assistants at least sometimes, and more than half (10, 55.6%) reported using them very often. In contrast, 11 participants (61.1%) reported no familiarity with web accessibility concepts such as WCAG or ARIA, and only two participants (11.1% combined) reported moderate or high familiarity.

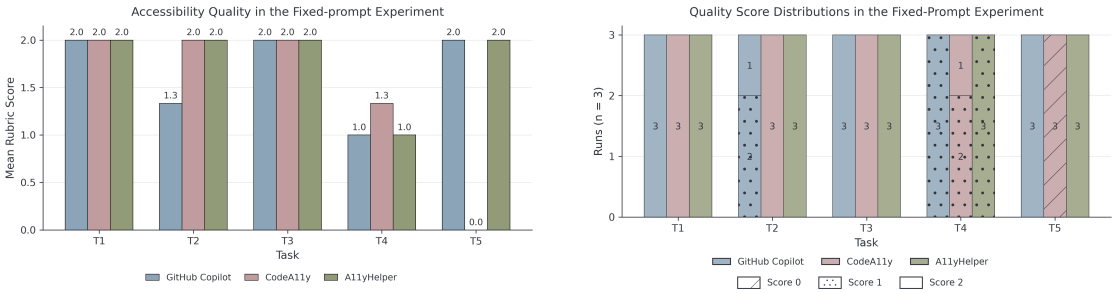
5.2 RQ1: Accessibility Quality

5.2.1 Fixed-Prompt Experiment

Figure 5.1 presents the results in the fixed-prompt experiment, averaged over three independent runs per tool-task pair. Panel (a) shows the mean scores, while panel (b) shows the score distribution. For T1 and T3, all three tools receive a score of 2 in every run.

Table 5.1: Participant background ($N = 18$)

Variable	Summary
Current status	Master student: 13 (72.2%); PhD student: 2 (11.1%) Bachelor student: 1 (5.6%); Industry professional: 1 (5.6%) Backend developer: 1 (5.6%)
Web development experience	Less than 1 year: 8 (44.4%); 1–2 years: 5 (27.8%) 3–5 years: 4 (22.2%); More than 5 years: 1 (5.6%)
HTML familiarity	2.89 ± 1.28
CSS familiarity	2.72 ± 1.49
JavaScript familiarity	2.67 ± 1.50
AI coding assistant use	Very often: 10 (55.6%); Sometimes: 4 (22.2%); Often: 4 (22.2%)
Accessibility familiarity	Not familiar at all: 11 (61.1%); Slightly familiar: 5 (27.8%) Moderately familiar: 1 (5.6%); Very familiar: 1 (5.6%)



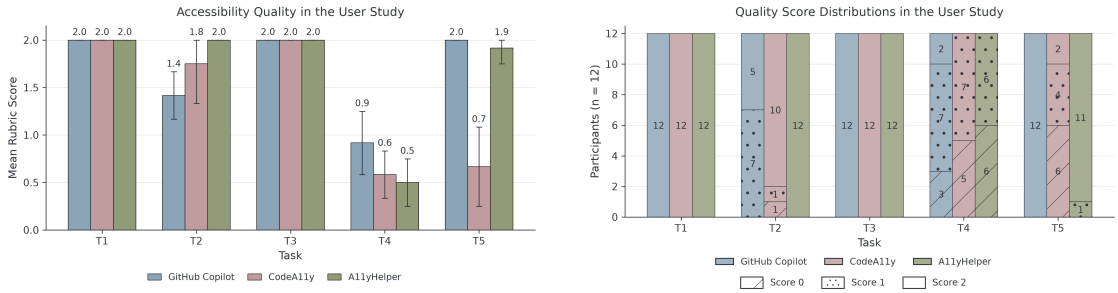
(a) Mean scores.

(b) Distribution of rubric scores.

Figure 5.1: Accessibility quality scores for the fixed-prompt experiment: (a) mean rubric scores; (b) score distributions.

For T2, all three CodeA11y outputs and all three A11yHelper outputs received a score of 2. The scores of GitHub Copilot were 2, 1 and 1 respectively, with an average of 1.3. For T4, CodeA11y had the highest observed mean of 1.3, with one output scoring 2 and two scoring 1. All outputs of GitHub Copilot and A11yHelper scoring 1, giving both tools a mean of 1.0. No output in T4 got 0 points.

The largest difference in mean scores appears in T5. GitHub Copilot and A11yHelper both receive 2.0, while CodeA11y receives 0.0. All three CodeA11y outputs receive Unacceptable ratings under the task-specific criteria, while all three outputs from each of the other two tools receive Good ratings.



(a) Mean scores.

(b) Distribution of rubric scores.

Figure 5.2: Accessibility quality scores for the user study: (a) mean rubric scores with bootstrap 95% confidence intervals; (b) distribution of rubric scores.

5.2.2 Controlled User Study

Figure 5.2 presents the user-study results from 12 final code artifacts per tool-task pair. Panel (a) shows the mean rubric scores with bootstrap 95% confidence intervals, while panel (b) shows the score distributions. T1 and T3 remain at the maximum mean score of 2.0 across all three tools, reproducing the pattern observed in the fixed-prompt experiment.

For T2, A11yHelper receives the maximum mean score of 2.0, CodeA11y produced ten artifacts scoring 2, one scoring 1, and one scoring 0, with a mean score of 1.8. GitHub Copilot produced five artifacts scoring 2 and seven scoring 1, producing a mean score of 1.4. A11yHelper is the only tool with a mean score of 2.0 for this task in both evaluations.

T4 again has the lowest mean scores among the five tasks. GitHub Copilot, CodeA11y, and A11yHelper receive 0.9, 0.6, and 0.5. Two GitHub Copilot artifacts scored 2, while no CodeA11y or A11yHelper artifact received this score.

For T5, all 12 GitHub Copilot artifacts scored 2. A11yHelper produced eleven artifacts scoring 2 and one scoring 1, producing a mean score of 1.9. CodeA11y produced two artifacts scoring 2, four scoring 1, and six scoring 0, producing a mean score of 0.7.

5.3 RQ2: Usability and Preference

5.3.1 SUS Scores

Table 5.2: Summary statistics of SUS scores by tool ($N = 12$ per tool)

Tool	Mean	Median	SD	95% CI
GitHub Copilot	79.38	78.75	13.28	[72.29, 86.46]
CodeA11y	59.58	53.75	15.95	[51.25, 68.54]
A11yHelper	76.46	76.25	12.18	[69.79, 83.12]

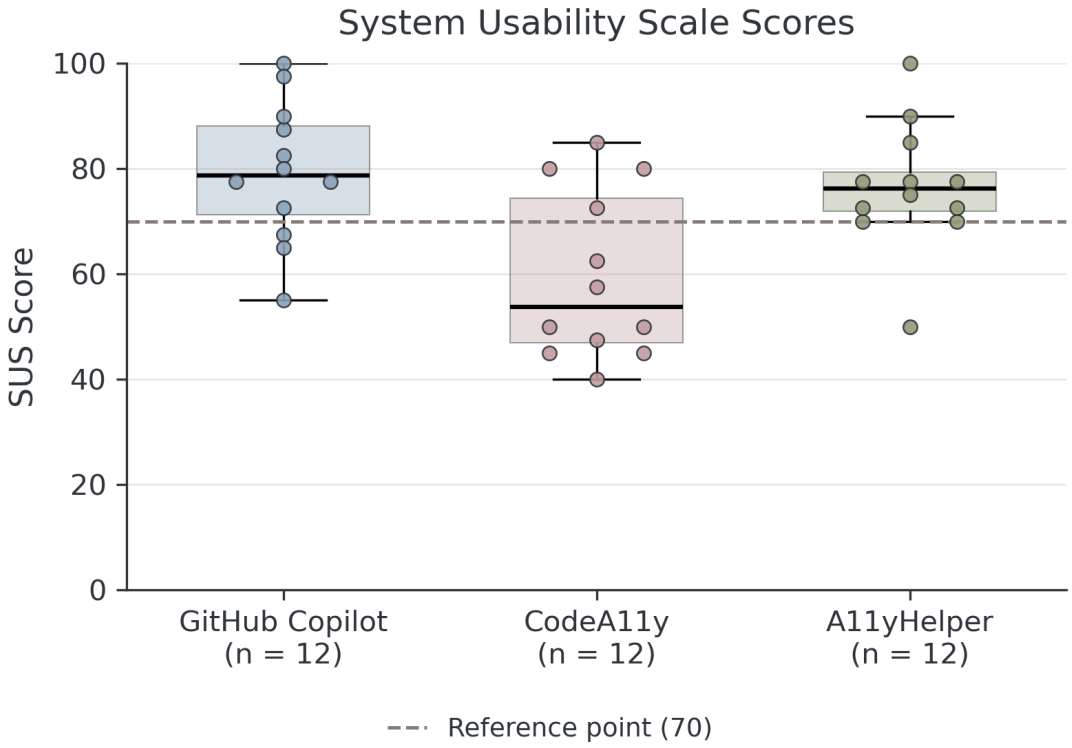


Figure 5.3: Distribution of SUS scores for each tool, with $N = 12$ ratings per tool. The dashed line marks 70 as an interpretive reference point [6].

Table 5.2 and Figure 5.3 report the SUS scores from 12 participant evaluations per tool. Figure 5.3 includes a score of 70 as an interpretive reference point [6]. Bangor et al. suggest that systems scoring above 70 can generally be considered acceptable, whereas scores below 70 may require further scrutiny and improvement [6].

GitHub Copilot has the highest mean SUS score of 79.38, followed by A11yHelper at 76.46. Both mean scores are above the reference point of 70. CodeA11y has a lower mean score of 59.58, which falls below this reference point.

CodeA11y has the largest standard deviation at 15.95, compared with 13.28 for GitHub Copilot and 12.18 for A11yHelper. Its SUS scores therefore show greater dispersion around the mean within this sample.

5.3.2 Tool Preference

After completing both assigned tool conditions, participants selected the tool they preferred overall and provided a written reason. Each participant chose only between the two tools they had used, and each tool was available for selection by 12 participants. Figure 5.4 shows that GitHub Copilot was selected by 8 of 18 participants (44.4%), followed by A11yHelper with 7 (38.9%) and CodeA11y with 3 (16.7%).

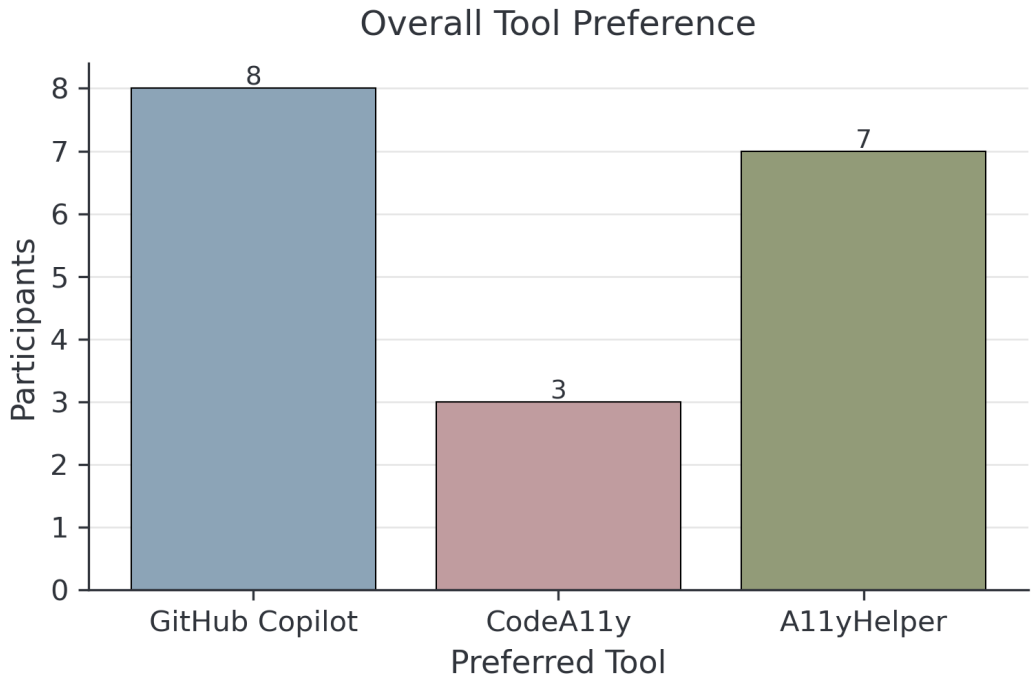


Figure 5.4: Overall tool preference ($N = 18$).

All 18 participants provided a written reason. Table 5.3 summarizes the descriptive categories, with each response assigned to one category.

Table 5.3: Descriptive categories of preference reasons.

Reason category	**GitHub Copilot**	**CodeA11y**	**A11yHelper**
Ease of use and IDE integration	7	0	1
Response clarity and organization	0	1	3
Generated results and requirement fit	1	1	1
Response speed and conciseness	0	1	2
Total	8	3	7

Seven of the eight responses favoring GitHub Copilot were categorized under ease of use and IDE integration. Participants value the convenience of applying changes and felt that fewer additional interactive steps were needed. Another feedback emphasized the consideration of edge cases and that the change was limited to the scope of stated requirements, despite the participant also mentioned that this would increase the cognitive load.

Among the seven feedbacks favoring A11yHelper, three were categorized under response clarity and organization. These feedbacks valued its concise accessibility

summary, integrated solutions and checks, and fewer running logs. In addition, the participants also mentioned that they had a greater sense of control over the suggested changes.

The three feedbacks supporting CodeA11y focus on different aspects. One participant valued the explanations and instructions about where to use the generated code. Another participant valued that the tool could closely follow the task requirements, reduce unnecessary code additions and fix errors. The third participant valued faster generation and could provide concise answers to simple development tasks.

6

Discussion

6.1 Interpretation of Findings

6.1.1 RQ1: Accessibility Quality

The value of accessibility-focused support varied across tasks. No tool achieved higher mean scores across all tasks in both evaluations. GitHub Copilot also received higher user-study mean scores on T1–T4 than those reported in the CodeA11y study. Advances in the underlying models are one possible explanation for this difference, but differences in participants, prompts, and study conditions prevent attributing it to model improvements alone.

For T1 and T3, all three tools achieved the highest score in both evaluations. These tasks required relatively localized changes, and the relevant information has been provided in the task description and surrounding code. This may explain why all three tools can meet the requirements. However, the ceiling scores prevented these tasks from distinguishing between tools. The third-level rubric cannot capture differences between solutions that met the same criteria, and the limited number of runs and participants may also lead to some occasional output changes not being captured.

T2 required both form labeling and keyboard navigation. All A11yHelper artifacts received Good ratings in both evaluations. CodeA11y received the same ratings in the fixed-prompt experiment, but two of its final artifacts in the user study were rated below Good. GitHub Copilot had lower mean scores in both evaluations. Among the five tasks, T2 showed the largest mean score difference between A11yHelper and GitHub Copilot.

T4 received low scores across all three tools, but their relative rankings differed between the two evaluations. In the fixed-prompt experiment, CodeA11y's highest mean score came from one Good output and two Average outputs, while each of the other tools produced three Average outputs. However, this difference was only based on three runs per tool and did not persist in the user study, where GitHub Copilot had the highest mean score, followed by CodeA11y and A11yHelper.

The user study allowed participants to write requests, ask follow-up questions, and

revise suggested code. None of the tools received image content directly, so alt-text generation relied on the available code context. During the study process, the researcher observed that some participants did not check or revise the generated alt text despite reminders. This may help explain some low-scoring outputs, but it does not establish why CodeA11y's relative ranking changed between the evaluations.

T5 requires the new Bluesky community card must follow the identification and structure of sibling cards elsewhere in the file. GitHub Copilot and A11yHelper received high scores in both evaluations, whereas CodeA11y received lower scores. This pattern is consistent with the CodeA11y authors' reported limitations in collecting code context around the cursor.

6.1.2 RQ2: Usability and Tool Preference

A11yHelper achieved a mean SUS score of 76.46, close to GitHub Copilot's 79.38, while CodeA11y received a lower mean of 59.58. CodeA11y also had the largest standard deviation, indicating greater dispersion around its mean within this sample. The difference between the two accessibility-focused tools suggests that how developers interact with specialized guidance matters alongside the functionality provided.

The preference counts followed the same descriptive ordering as the SUS scores. The reasons for the choice given by the participants offer possible explanations for these evaluations. Among the eight feedbacks that selected GitHub Copilot, seven mentioned ease of use and IDE integration. Participants mentioned convenient application of changes and a perceived need for fewer interaction steps. Participants who choose A11yHelper value a clear answer structure, a concise summary of accessible questions, and a feeling of faster response. The three participants who chose CodeA11y paid different attention to different aspects, namely the explanations of the generated code, requirement-focused results, and perceived speed and conciseness. These feedbacks show that developers not only value the accessibility-related guidance, but also value whether the suggestion is easy to understand and to be applied to actual development.

Together, the findings show that accessibility quality and developer experience both matter in evaluating these assistants. GitHub Copilot had the highest mean SUS score and the largest preference count, but did not receive the highest quality scores on every task. CodeA11y received comparatively high quality scores on T2 despite its lower mean SUS score. These patterns show that quality scores alone provide an incomplete basis for assessing an assistant's practical value. Evaluating accessibility assistants therefore requires attention to how developers experience and apply the guidance, alongside the task-level accessibility quality of the resulting code.

6.2 Limitations

A11yHelper uses TODO and FIXME comments to mark the content that requires developers to supplement information or verify, but these reminders still depend on the developer to complete the follow-up check. In the user study, some participants skipped checking the generated alternative text despite seeing these comments. The CodeA11y study reported a related difficulty, participants showed little interest in floating popups and later began closing modal reminders [37]. These observations suggest that reminders can draw attention to unresolved content without ensuring that developers verify it. Therefore, for content that requires manual verification, A11yHelper still relies on developers to follow up.

A11yHelper's `/scan` workflow is limited by the rule coverage of the underlying axe scanner. Restructuring scan results and providing LLM-based explanations do not expand that coverage. Problems outside the scanner's rules may therefore be absent from the detected issue list. Scan-based assistance can support remediation of detected violations, but it cannot replace a broader accessibility review.

A11yHelper also inherits limitations from the underlying LLM. Although its prompts provide accessibility-specific instructions, the model may still generate incorrect or incomplete suggestions that do not fully satisfy accessibility requirements. Its outputs may also vary across repeated requests, even when the input remains unchanged. Therefore, A11yHelper cannot guarantee the correctness or accessibility of generated code, and developers must still review and validate its suggestions.

6.3 Threats to Validity

Several factors may affect the interpretation and generalizability of the findings. This section discusses threats to external, internal, construct, and conclusion validity.

6.3.1 External Validity

One threat to external validity concerns the participant sample. The user study included 18 participants, the small, student-heavy sample limits the diversity of developer backgrounds represented. Table A.2 shows higher mean SUS scores for participants with three or more years of experience, but no consistent pattern in accessibility quality scores across tasks. This group included only five participants, limiting conclusions about the relationship between web development experience, perceived usability, and accessibility quality. Senior developers may have different expectations of accessibility guidance and different ways of reviewing and applying generated suggestions.

A second threat concerns the range of experimental settings. The five tasks were selected from two open-source repositories, providing realistic development contexts but covering only a small subset of project structures and technology stacks. These choices broadened the evaluated requirements but did not establish coverage of web

development tasks in general. Performance may differ in projects with other architectures, frameworks, or development conventions. Evaluation across more tasks and codebases is needed to assess whether the observed patterns extend beyond similar settings.

6.3.2 Internal Validity

The number of repetitions of the fixed-prompt experiment is limited, and it is difficult to assess the stability of the observed tool performance. Due to the randomness of the LLM output, LLMs may generate different code even when given identical prompts. In addition, the GitHub Copilot Chat interface did not provide options to control temperature or random seeds, so this study did not control these random factors. Each tool was run independently only three times on each task, and the results were not enough to fully reflect the distribution of accessibility quality scores of the generated code. The fixed-prompt experiment provides a limited comparison under the same prompt conditions, and the results should be used as supplementary evidence for the controlled user study.

During the user study, participants may have inferred the accessibility focus of the study before completing the tasks. The background questionnaire asked about familiarity with concepts such as WCAG and ARIA, which may have increased participants' attention to accessibility during task execution. To reduce influence, the experimental instructions given to participants did not have any accessibility guidance or task-specific hints. Nevertheless, the potential priming effect of the questionnaire cannot be fully excluded.

Another threat concerns model changes during the user study. During the user study, the model in GitHub Copilot Chat was changed from GPT-4.1 to GPT-5.5, which was introduced by the platform. Nine participants used GPT-4.1 and nine participants used GPT-5.5. For each participant, A11yHelper, CodeA11y and GitHub Copilot all ran under the same model. Table A.3 shows higher mean SUS scores in the GPT-5.5 subgroup than in the GPT-4.1 subgroup for all three tools. Differences in accessibility quality scores varied by tool and task, with no consistent pattern. Although each tool had equal numbers of participants in the two model subgroups, model version may still have influenced the tool comparisons.

Tool order and the learning effect are another threat affecting the internal validity. The experience gained by participants when using the first tool may make them more familiar with the code repository, task requirements or feasible solutions, thus affecting their performance when using the second tool. To reduce this impact, this study adopts BIBD and balances the tool order among subgroups. This design reduces the systematic order imbalance, but it does not completely eliminate the learning effect from repeating the tasks.

6.3.3 Construct Validity

A solution rated Good may still have accessibility problems outside the properties covered by the task-specific rubric. The evaluation used predefined criteria representing selected accessibility requirements at three quality levels. However, these criteria do not cover every relevant property of the resulting interface. A tool could satisfy the task requirements while leaving other accessibility issues unresolved or introducing issues outside the rubric's scope. The accessibility quality scores therefore indicate how well solutions met the evaluated requirements, rather than establishing that they were fully accessible.

The second threat is that the evaluation of the complete coding assistant cannot distinguish the independent effects of individual features. This study compares complete coding assistants and allows participants to freely interact with the tools assigned to them in user study. The study did not record which Handlers the participants used A11yHelper, and interaction counts, suggestion acceptance, or chat logs. Consequently, the results cannot establish how often features such as `/scan` were used or determine the individual contributions of scanning, request routing, or context construction to code quality. Similarly, participants' comments on speed and interaction steps with fewer reflect their subjective feelings, rather than measured performance differences.

Task selection also introduces another potential bias. T5 was designed in this study to evaluate a task requiring information distributed across a larger file, which is in line with the context construction strategy of A11yHelper. Although T5 is based on the real Kubernetes issue and follows the task selection and evaluation approach of previous study, its design may be more conducive to tools that can access the relevant context in the file. The contribution of context construction to the observed differences could not be isolated without an ablation study.

Another threat is the ceiling effect observed in T1 and T3. All three tools got the maximum score in both evaluations, limiting the ability of these tasks to distinguish the performance of tools. The task-specific rubric used three quality levels, and could not capture the difference between solutions that meet the highest level criteria. Therefore, although the results of T1 and T3 prove that all three tools meet the requirements, their scores provide limited evidence for tool performance comparison.

6.3.4 Conclusion Validity

Before scoring, the researcher assigned anonymous record numbers and hides the participant's identity and tool conditions. Researchers and thesis supervisors review the scoring results together according to the task-specific rubric. However, this study did not arrange the second evaluator to score independently, so inter-rater agreement could not be assessed. Therefore, the consistency of scoring judgments among different independent evaluators remains uncertain, and some scoring results may be affected by the subjective interpretation of the rubric.

Differences in the participants who evaluated each tool may also affect the comparison of overall scores. The 12 participants were not the same for every tool. Each pair of tools shared six participants who had used both, while the other participants differed. Although the design balanced the number of evaluations and tool order, differences between participants may still have contributed to the overall quality and SUS means for each tool.

6.4 Future Work

Future work could extend A11yHelper's visual and code context, improve support for developer verification, and evaluate outcomes beyond the current task scores.

One technical research direction is to extend the context construction beyond the current active file. A feasible solution is to retrieve the relevant files required for the current task along the import relationship and component references. Further extensions can use repository-level retrieval of shared components, styles, and other relevant project information. Comparing context strategies and parameter settings would help determine whether the additional information improves accessibility quality and how it affects response time and input size.

Future work could also make unresolved TODO and FIXME markers easier to review, for example by listing them as editor diagnostics. The user-study observations suggest that annotations alone did not ensure alt-text verification. An evaluation could examine whether more visible prompts encourage developers to complete these checks and whether the prompts interrupt their work.

The code artifacts collected in this study can be re-evaluated by accessibility experts. By comparing the expert score with the original score, the difference between the two in the evaluation results can be identified. Future study can also introduce more challenging accessibility tasks or adopt more detailed scoring criteria to improve the ability to distinguish the assessment. In this way, even if the solutions reach the highest level of existing scoring standards, they can further identify their differences in accessibility quality.

Future evaluations could collect a broader range of data and examine longer-term outcomes. Accessibility experts could provide more detailed assessments of generated code, while testing the resulting interfaces with users of assistive technologies could reveal whether the changes support successful interaction and which barriers remain. A longitudinal study in development teams could examine whether developers continue to use A11yHelper's feedback and how they incorporate it into accessibility decisions over time. Recording developer interactions and workflows would also help relate their experiences and outcomes to specific forms of assistance.

6.5 Use of Generative AI Tools

The author used generative AI tools to suggest grammar corrections to written text and to assist with LaTeX conversion in parts of the thesis. The author reviewed all suggested corrections and formatting changes before incorporating them. The Chalmers policy on generative AI use in thesis preparation provides the relevant guidance [12].

7

Conclusion

This thesis investigated how an accessibility-focused coding assistant compares with existing assistants in task-level accessibility quality and perceived usability. It presented A11yHelper, a VS Code extension that integrates accessibility guidance, axe scans, and and file-size-aware code context into GitHub Copilot Chat. We compared it with GitHub Copilot and CodeA11y in a fixed-prompt experiment and in a balanced incomplete block user study with 18 participants, using five tasks and blind scoring against task-specific rubrics.

The results show tool differences varied by task. For the two localized tasks, all three tools met the requirements in both evaluations, so the task ratings showed no advantage for the accessibility-specific tools. Differences appeared on the task combining form labeling and keyboard navigation, where A11yHelper had a mean of 2.0 in both evaluations against 1.3 and 1.4 for GitHub Copilot. For the task requiring information from different parts of a file, CodeA11y scored 0.0 and 0.7, while GitHub Copilot and A11yHelper scored 1.9 or higher. Alternative text received the lowest mean scores across tools. Only two of the 36 user-study artifacts received Good ratings, both from GitHub Copilot. The researcher also observed that some participants did not verify the generated alternative text despite TODO reminders.

The results did not show an overall accessibility quality advantage for A11yHelper over GitHub Copilot. GitHub Copilot matched or exceeded A11yHelper's mean quality scores on every task except the task combining form labeling and keyboard navigation. It also had the highest mean SUS score at 79.38, followed by A11yHelper at 76.46 and CodeA11y at 59.58, and received the most preference selections: eight, compared with seven and three, respectively. CodeA11y had higher mean quality scores than GitHub Copilot on the form task in both evaluations but a lower mean SUS score. The tool ordering by task-level accessibility quality therefore did not consistently match the ordering by perceived usability.

This thesis contributes A11yHelper as an implementation of accessibility support within GitHub Copilot Chat. Its comparative evaluation identifies task-specific strengths and limitations alongside developers' usability assessments and preferences. Together, these contributions provide an implementation to build on and empirical evidence to guide further evaluation. Future work should investigate visual and cross-file context, support for reviewing unresolved accessibility issues, and

ablation studies to assess the contributions of individual features.

Bibliography

[1] Web accessibility directive — standards and harmonisation. https://digital-strategy.ec.europa.eu/en/policies/web-accessibility-directive-standards-and-harmonisation. Accessed: 2026-03-30.

[2] Web content accessibility guidelines (wcag) 2.1. https://www.w3.org/TR/WCAG21/, 2026.

[3] Zahra Shakeri Hossein Abad, Oliver Karras, Kurt Schneider, Ken Barker, and Mike Bauer. Task interruption in software development projects: What makes some interruptions more disruptive than others? In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 122–132, 2018.

[4] Siddikjon Gaibullojonovich Abduganiev. Towards automated web accessibility evaluation: a comparative study. *International Journal of Information Technology and Computer Science*, 9(9):18–44, 2017.

[5] Abdullah Alsaeedi. Comparing web accessibility evaluation tools and evaluating the accessibility of webpages: proposed frameworks. *Information*, 11(1):40, 2020.

[6] Aaron Bangor, Philip T Kortum, and James T Miller. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, 24(6):574–594, 2008.

[7] BBC. bbc/simorgh. https://github.com/bbc/simorgh. GitHub repository. Accessed: 2026-03-27.

[8] Tingting Bi, Xin Xia, David Lo, John Grundy, Thomas Zimmermann, and Denae Ford. Accessibility in software practice: A practitioner's perspective. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4):1–26, 2022.

[9] Raj Chandra Bose. On the construction of balanced incomplete block designs. *Annals of Eugenics*, 9(4):353–399, 1939.

[10] John Brooke. Sus: a "quick and dirty'usability. *Usability evaluation in industry*,

189(3):189–194, 1996.

[11] Milton Campoverde-Molina, Sergio Luján-Mora, and Llorenç Valverde. Process model for continuous testing of web accessibility. *IEEE access*, 9:139576–139593, 2021.

[12] Chalmers University of Technology. Master's thesis in software engineering: Generative AI policy. `https://canvas.chalmers.se/courses/16233/pages/masters-thesis-in-software-engineering#genai`. Accessed: 2026-10-05.

[13] Chrome for Developers. Lighthouse accessibility score. `https://developer.chrome.com/docs/lighthouse/accessibility/scoring`. Accessed: 2026-02-17.

[14] Deque Labs. axe-core/lib/rules/no-naming-method-matches.js at 0d4e4e70aa9f46519eb6000744e043c058fd994e. `https://github.com/dequelabs/axe-core/blob/0d4e4e70aa9f46519eb6000744e043c058fd994e/lib/rules/no-naming-method-matches.js`. GitHub. Accessed: 2026-02-17.

[15] Deque Systems. axe-core issue impact. `https://github.com/dequelabs/axe-core/blob/develop/doc/issue_impact.md`. Official documentation.

[16] Muhammad Arif Faizin, Muhammad Nevin, and Umi Laili Yuhana. Indonesia e-government website performance and accessibility evaluation using automated tool lighthouse. In *2024 2nd International Conference on Software Engineering and Information Technology (ICoSEIT)*, pages 210–215. IEEE, 2024.

[17] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023.

[18] GitHub. Github copilot · your ai pair programmer. `https://github.com/features/copilot`. Accessed: 2026-03-09.

[19] GitHub. GitHub Copilot Features, 2024. Accessed: 2026-03-30.

[20] Carlos Adriano Gonçalves and Célia Talma Gonçalves. Assessment on the effectiveness of github copilot as a code assistance tool: an empirical study. In *EPIA conference on artificial intelligence*, pages 27–38. Springer, 2024.

[21] Ziyao He, Syed Fatiul Huq, and Sam Malek. Enhancing web accessibility: Automated detection of issues with generative ai. *Proceedings of the ACM on Software Engineering*, 2(FSE):2264–2287, 2025.

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

pages 1–4, 2014.

[23] Kubernetes. kubernetes/website. `https://github.com/kubernetes/website`. GitHub repository. Accessed: 2026-03-27.

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

[25] Shashank Kumar, Jeevitha Shree DV, and Pradipta Biswas. Comparing ten wcag tools for accessibility evaluation of websites. *Technology and Disability*, 33(3):163–185, 2021.

[26] Zui Young Lim, Jia Min Chua, Kaiting Yang, Wei Shin Tan, and Yinn Chai. Web accessibility testing for singapore government e-services. In *Proceedings of the 17th International Web for All Conference*, pages 1–5, 2020.

[27] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173, 2024.

[28] Juan-Miguel López-Gil and Juanan Pereira. Turning manual web accessibility success criteria into automatic: an llm-based approach. *Universal Access in the Information Society*, 24(1):837–852, 2025.

[29] Kelly Mack, Edward Cutrell, Bongshin Lee, and Meredith Ringel Morris. Designing tools for high-quality alt text authoring. In *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*, pages 1–14, 2021.

[30] Gloria Mark, Daniela Gudith, and Ulrich Klocke. The cost of interrupted work: more speed and stress. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 107–110, 2008.

[31] Forough Mehralian, Titus Barik, Jeff Nichols, and Amanda Swearngin. Automated code fix suggestions for accessibility issues in mobile apps. *arXiv preprint arXiv:2408.03827*, 2024.

[32] Microsoft. Ai extensibility in vs code. `https://code.visualstudio.com/api/extension-guides/ai/ai-extensibility-overview`. Accessed: 2026-02-18.

[33] Microsoft. Common capabilities. `https://code.visualstudio.com/api/extension-capabilities/common-capabilities`. Accessed: 2026-02-18.

[34] Microsoft. Extension host. `https://code.visualstudio.com/api/advanced-topics/extension-host`. Accessed: 2026-02-18.

[35] Microsoft. Setting up Visual Studio Code. `https://code.visualstudio.com/docs/setup/setup-overview`. Accessed: 2026-02-18.

[36] Peya Mowar. Accessibility in ai-assisted web development. In *Proceedings of the 21st International Web for All Conference*, pages 123–125, 2024.

[37] Peya Mowar, Yi-Hao Peng, Jason Wu, Aaron Steinfeld, and Jeffrey P Bigham. Codea11y: Making ai coding assistants useful for accessible web development. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2025.

[38] PD Parthasarathy and Swaroop Joshi. Accessibility education for software engineers: Evaluating the impact of game-based learning. *ACM Transactions on Computing Education*, 26(2):1–47, 2026.

[39] Helen Petrie, Andreas Savva, and Christopher Power. Towards a unified definition of web accessibility. In *Proceedings of the 12th International Web for All Conference*, pages 1–13, 2015.

[40] 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*, pages 1–4, 2022.

[41] Mohaimenul Azam Khan Raiaan, Md Saddam Hossain Mukta, Kaniz Fatema, Nur Mohammad Fahad, Sadman Sakib, Most Marufatul Jannat Mim, Jubaer Ahmad, Mohammed Eunus Ali, and Sami Azam. A review on large language models: Architectures, applications, taxonomies, open issues and challenges. *IEEE access*, 12:26839–26874, 2024.

[42] Abdul Razzaq, Jim Buckley, Qin Lai, Tingting Yu, and Goetz Botterweck. A systematic literature review on the influence of enhanced developer experience on developers' productivity: Factors, practices, and recommendations. *ACM Computing Surveys*, 57(1):1–46, 2024.

[43] Saima Ritonummi, Valtteri Siitonen, Markus Salo, Henri Pirkkalainen, and Anu Sivunen. Flow experience in software engineering. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 618–630, 2023.

[44] Viktoria Stray, Nils Brede Moe, Nivethika Ganeshan, and Simon Kobbenes. Generative ai and developer workflows: How github copilot and chatgpt influence solo and pair programming. In *Proceedings of the 58th Hawaii International Conference on System Sciences*, 2025.

[45] U.S. Department of Justice. Americans with disabilities act title ii regulations, 2010. Accessed: 2026-03-30.

[46] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems*

extended abstracts, pages 1–7, 2022.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[48] Sanni-Anni Vuorinen. Accessibility testing in software development. 2023.

[49] Andrew Walker, Michael Coffey, Pavel Tisnovsky, and Tomas Cerny. On limitations of modern static analysis tools. In *Information Science and Applications: ICISA 2019*, pages 577–586. Springer, 2019.

[50] WebAIM. The WebAIM million: The 2026 report on the accessibility of the top 1,000,000 home pages. `https://webaim.org/projects/million/`, March 2026. Last updated March 30, 2026. Accessed September 30, 2026.

[51] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[52] World Wide Web Consortium (W3C). Web Content Accessibility Guidelines (WCAG) 2.0. Technical report, W3C, 2008. W3C Recommendation, 11 December 2008. Accessed: 2026-03-30.

[53] World Wide Web Consortium (W3C). Understanding success criterion 3.2.4: Consistent identification. `https://www.w3.org/WAI/WCAG21/Understanding/consistent-identification.html`, 2018. WCAG 2.1 Understanding Documents.

[54] World Wide Web Consortium (W3C). Web Content Accessibility Guidelines (WCAG) 2.2. Technical report, W3C, 2023. W3C Recommendation, 5 October 2023. Accessed: 2026-03-30.

[55] Frank Yates. Incomplete randomized blocks. *Annals of eugenics*, 7(2):121–140, 1936.

[56] Yeliz Yesilada and Simon Harper. Web accessibility. *Human–Computer Interaction Series. Springer London*, 2019.

[57] Holly Yu. Web accessibility and the law: recommendations for implementation. *Library Hi Tech*, 20(4):406–419, 2002.

[58] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Guoyin Wang, et al. Instruction tuning for large language models: A survey. *ACM Computing Surveys*, 58(7):1–36, 2026.

[59] Yuxin Zhang, Sen Chen, Lingling Fan, Chunyang Chen, and Xiaohong Li. 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*, pages 1255–1267, 2023.

A

Appendix

A.1 Fixed Prompts

Table A.1: Fixed prompts used for the five evaluation tasks

Task	Prompt
T1	Add a new "Learn More" button next to the existing Quickstart button and make it visually consistent with the current page design.
T2	Add a KubeWeekly newsletter subscription section with an email input and a subscribe button that fits the existing page layout.
T3	Add a Top Stories section that displays the latest news articles from the existing data in a clean and readable list.
T4	Add a Sport section that displays the available sports articles as image cards, showing the article image, headline, and publication date.
T5	Add a Bluesky community card by following the structure and identification pattern of existing community resource cards.

A.2 Consent Form

Introduction

You are invited to participate in a research study conducted at Chalmers University of Technology. The study investigates how AI coding assistants support developers during web development tasks.

During the session, you will complete a set of frontend development tasks using two different AI tools, and fill in short questionnaires. The session will take approximately **60–90** minutes.

Participation is entirely voluntary. You may withdraw at any time without consequence.

Data We Collect

- **Task results** — the code you produce during the tasks
- **Questionnaire responses** — your ratings and written feedback about the tools

Data Protection

All data will be anonymized before analysis. Your name will not be associated with any results in publications or reports. Data will be stored securely and only accessed by the research team.

This study is conducted in accordance with GDPR and Chalmers University of Technology's data protection policy.

Contact Information

If you have any questions about this study, please contact:

Dantong Luo

`dantong@chalmers.se`

Informed Consent Form

I have read and understood the information above.

I understand that my participation is voluntary and that I can withdraw at any time.

I agree that my task results and questionnaire responses may be used for research purposes.

Participant Name*

Participant Email*

A.3 Background Survey

This survey collects basic information about your background and experience in web development and AI tools.

1. **What is your current status?***
    - Bachelor student
    - Master student

- PhD student
- Industry professional
- Other:

2. **How many years of experience do you have in web development?***
   - Less than 1 year
   - 1–2 years
   - 3–5 years
   - More than 5 years
3. **How familiar are you with the following technologies? (1 = Not familiar, 5 = Very familiar)***

	1	**2**	**3**	**4**	**5**
HTML	☐	☐	☐	☐	☐
CSS	☐	☐	☐	☐	☐
JavaScript	☐	☐	☐	☐	☐

4. **How often do you use AI coding assistants (e.g., GitHub Copilot, ChatGPT)?***
   - Never
   - Rarely
   - Sometimes
   - Often
   - Very often
5. **Which AI tools have you used before? (Select all that apply)**
   - GitHub Copilot
   - ChatGPT
   - Gemini
   - Claude
   - Other:
6. **How familiar are you with web accessibility concepts (e.g., WCAG,**

ARIA)?*

- Not familiar at all
- Slightly familiar
- Moderately familiar
- Very familiar

A.4 System Usability Survey

This questionnaire evaluates your experience with the AI coding assistant you just used.

1. **Which tool did you use for the tasks?***
   - A11yHelper
   - CodeA11y
   - GitHub Copilot
2. **I think that I would like to use this system frequently.***

   Strongly disagree 1 2 3 4 5 Strongly agree
3. **I found the system unnecessarily complex.***

   Strongly disagree 1 2 3 4 5 Strongly agree
4. **I thought the system was easy to use.***

   Strongly disagree 1 2 3 4 5 Strongly agree
5. **I think that I would need the support of a technical person to be able to use this system.***

   Strongly disagree 1 2 3 4 5 Strongly agree
6. **I found the various functions in this system were well integrated.***

   Strongly disagree 1 2 3 4 5 Strongly agree
7. **I thought there was too much inconsistency in this system.***

   Strongly disagree 1 2 3 4 5 Strongly agree
8. **I would imagine that most people would learn to use this system very quickly.***

Strongly disagree 1 2 3 4 5 Strongly agree

9. **I found the system very cumbersome to use.***

   Strongly disagree 1 2 3 4 5 Strongly agree

10. **I felt very confident using the system.***

    Strongly disagree 1 2 3 4 5 Strongly agree

11. **I needed to learn a lot of things before I could get going with this system.***

    Strongly disagree 1 2 3 4 5 Strongly agree

12. **Have you completed tasks using all tools?***

    If you have only completed one tool, select "No".

    If you have finished the tasks with both tools, select "Yes".

    - No
    - Yes

Additional Questions (After Using All Tools)

Which tool do you prefer overall? *

- A11yHelper
- CodeA11y
- GitHub Copilot

Why do you prefer this tool? *

A.5 User Study Guide: AI-Assisted Web Development

Welcome

Thank you for participating in this study. This session will take approximately **60–90 minutes**.

You will complete a set of web development tasks using AI coding tools. There are no right or wrong answers. **We are evaluating the tools, not you.** Please work naturally, as you would in your daily development routine.

What You Will Do

You will first complete a short **consent and background survey**, followed by a brief **introduction** to the tools.

Then, you will work on **five development tasks using Tool A** and fill in a short **questionnaire** about your experience.

After that, you will **complete the same five tasks using Tool B**, followed by **another questionnaire**.

> **Note:** The tasks are identical for both tools. Within each group, the order of tools may vary between participants.

Development Tasks

You will complete **five tasks** in total, working across two real-world open-source projects.

Projects

Project	Preview URL	Task ID	Location (file path)
Kubernetes Website	`http://localhost:1313/`	T1	`layouts/page/home.html`
		T2	`content/en/_index.html`
		T5	`content/en/community/_index.html`
BBC Simorgh	`http://localhost:7080/`	T3	`src/app/components/MostRead/TopStories/index.tsx`
		T4	`src/app/components/MostRead/SeoImageGrid/index.tsx`

How to Find Each Task

Each task is marked with a comment `TODO: TASK Tn` in the source code.

Follow the description in the comment to complete the task. The comment will tell you exactly what to add or modify.

Using the AI Tool

The AI coding assistant is available in the **GitHub Copilot Chat panel** in the sidebar.

There are three tools you may use:

- For **A11yHelper**, you need to call it using `@a11yhelper` in your message.
- For **CodeA11y**, you need to call it using `@CodeA11y` in your message.
- For **GitHub Copilot**, you can just type your request directly in the chat.

For **A11yHelper** and **CodeA11y**, click the ***Apply in Editor*** button to apply the changes.

You are encouraged to use the AI tool as much as you like. There is no penalty for asking many questions or requesting multiple suggestions.

Rules During the Session

Allowed

- Using the assigned AI coding assistant.
- Searching online (e.g., Google, MDN) for general programming information.

Not Allowed

- Using other generative AI tools (e.g., ChatGPT, Claude, Gemini) outside of the assigned tool.
- Asking other people for help.

Getting Help from the Researcher

The researcher will **not provide hints or guidance on how to solve tasks**. However, if you experience any of the following, please raise your hand:

- The tool is not responding or behaving unexpectedly.
- There is an environment or setup issue.
- You are unsure which file to edit.

After All Tasks

Once you have completed all tasks:

1. Raise your hand to let the researcher know.
2. Complete the tool survey.

Questions Before We Begin?

If anything is unclear, please ask now. Once the session starts, the researcher will only be available for technical issues.

Thank you again for your time and participation.

A.6 User-Study Results by Web Development Experience

Table A.2: User-study results by web development experience.

Experience	Tool	n	T2	T4	T5	SUS
≤ 2 years	A11yHelper	9	2.00	0.56	1.89	73.6
	CodeA11y	8	1.62	0.62	0.62	58.4
	GitHub Copilot	9	1.44	1.00	2.00	74.7
≥ 3 years	A11yHelper	3	2.00	0.33	2.00	85.0
	CodeA11y	4	2.00	0.50	0.75	61.9
	GitHub Copilot	3	1.33	0.67	2.00	93.3

T1 and T3 are omitted because all artifacts scored 2.

A.7 User-Study Results by Model Version

Table A.3: User-study results by model version.

Tool	Model	n	T2	T4	T5	SUS
A11yHelper	GPT-4.1	6	2.00	0.50	1.83	73.3
	GPT-5.5	6	2.00	0.50	2.00	79.6
CodeA11y	GPT-4.1	6	1.67	0.67	0.83	56.2
	GPT-5.5	6	1.83	0.50	0.50	62.9
GitHub Copilot	GPT-4.1	6	1.33	1.00	2.00	78.3
	GPT-5.5	6	1.50	0.83	2.00	80.4

n denotes the number of participants evaluating each tool under each model version. T1 and T3 are omitted because all artifacts scored 2.