SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

From UI Images to Accessible Code: Leveraging LLMs for Automated Frontend Generation

Marco Lutz

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

From UI Images to Accessible Code: Leveraging LLMs for Automated Frontend Generation

Von UI-Bildern zu barrierefreiem Code: Nutzung von LLMs für die automatisierte Frontend-Generierung

Author:	Marco Lutz
Supervisor:	Sidong Feng
Advisor:	Prof. Dr. Chunyang Chen
Submission Date:	11.08.2025

I confirm that this bachelor's thesis in information systems is my own work and I have documented all sources and material used.

Munich, 11.08.2025 Marco Lutz

Acknowledgments

I would like to express my appreciation to Prof. Dr. Chunyang Chen for providing the opportunity to conduct this thesis within the Chair of Software Engineering & AI. I am especially grateful to Sidong Feng for his direct guidance, constructive feedback, and continuous support, which have been very valuable throughout the research and writing process.

Abstract

As Multimodal Large Language Models (MLLMs) increasingly support UI code generation from visual inputs (e.g., UI screenshots), their impact in accelerating frontend development is growing. While prior work has explored the generation of functional and visually accurate code, its accessibility remains less explored. This thesis investigated the accessibility of MLLM-generated HTML/CSS code from UI screenshots in an empirical study. We evaluate multiple state-of-the-art MLLMs with different prompting strategies across benchmark and real-world datasets. Our study investigates five research questions: (1) whether MLLMs can generate accessible code by default, (2) how model differences impact accessibility outcomes, (3) whether advanced prompting techniques improve accessibility, (4) how consistent accessibility violations are across different MLLMs given the same UI screenshot, and (5) the presence of potential data leakage in model training. Our findings show that even though MLLMs demonstrate high performance in code fidelity, they often fail to fulfill critical accessibility requirements. We highlight common violations, analyze prompting effects, and discuss implications for model training and evaluation. Based on our findings, we propose future research directions to enhance accessibility in AI-driven frontend development.

Contents

1 Introduction

1.1 Motivation

High-quality webpages are the backbone of our modern society. For billions of people, the internet and thus webpages are the central access point for information, education, work, trade, and culture. As of 2024, there are an estimated 1.1 billion active websites globally, with approximately 252,000 new sites launched each day [10].
Among the many attributes that define a successful website, *accessibility* stands out as a fundamental requirement. It ensures that individual users with visual, hearing, or cognitive impairments, as well as users of assistive technologies, can follow the content of a website. International standards, such as the Web Content Accessibility Guidelines (WCAG) [36], guide developers to build inclusive digital experiences by adhering to the official principles. According to WCAG, accessible web content must be perceivable, operable, understandable, and robust. Yet, despite its importance, accessibility is frequently overlooked in practice, resulting in persistent barriers for over one billion people globally who live with some form of disability [11].
The motivation for this rethinking is not purely technical but deeply ethical and societal. It is a legal requirement and a civil right in many jurisdictions, protected by laws such as the Americans with Disabilities Act (ADA) in the United States [34] and the more recent European Accessibility Act (EAA) in the EU [13]. Neglecting to follow these regulations could result in warnings, reputational damage and future sales loss.
At the same time, Large-Language Models (LLMs) have demonstrated significant improvements in code-related tasks, including code generation, completion and summarization. Current Tools, such as GitHub Copilot [17], Cursor [21], and Windsurf [22], are capable of supporting developers by generating functional code snippets from natural language descriptions. Recent developments in Multimodal Large Language Models (MLLMs) have further extended this capability to *Image-to-Code* tasks, where, based on a (UI) screenshot or design artifact, MLLMs generate functional HTML/CSS code. This workflow closely aligns with how developers and designers intuitively approach UI creation [7, 15]. This image-to-code principle significantly simplifies the front-end development process and reduces the need for manual markup creation. Based on this idea, an increasing number of research has explored techniques to improve UI code generation [6, 28, 38], leveraging MLLMs to better capture layout structures, semantics and component hierarchies.
While many MLLMs have demonstrated impressive capabilities in generating functional and visually accurate web UI code, their performance in generating accessible code remains unclear. This question has hardly been investigated to date. Aljedaani et al. [1] evaluated ChatGPT's capabilities to generate accessible websites based on natural language prompts provided by developers. Similarly, Suh et al. [32] compared LLM-generated code with human counterparts regarding accessibility compliance. However, these studies rely on natural language inputs, which do not reflect the real-world UI development workflows where visual UI designs (e.g. screenshots) serve as the primary input. Therefore, this thesis tries to close this gap by investigating the capabilities of MLLMs to generate accessible HTML/CSS code from visual web UI inputs.

1.2 Research Questions

This thesis investigates the following research questions:

RQ1: Do MLLMs generate accessible code from UI screenshots? This question investigates whether leading MLLMs, such as GPT-4o and Gemini 2.0 Flash, can generate accessible HTML/CSS code from UI screenshots sampled from real-world public datasets (Design2Code and WebCode2M). The experiment is based on a naive prompting strategy that requests code generation and does not explicitly instruct the model to prioritize accessibility. The generated code is then automatically evaluated through accessibility auditing tools and manual analysis to identify potential violations of the WCAG 2.1 guidelines. This question seeks to uncover whether current MLLMs inherently incorporate accessibility during code generation.

RQ2: Do different MLLMs vary in their ability to generate accessible UI code? To investigate how different models and their sizes affect the accessibility performance, this question compares the performance of a broader set of MLLMs, including both closed-source models (e.g., GPT-4o, Gemini 2.0 Flash) and open-source models (e.g., Qwen2.5-VL-7B, Llava-7B). Using the same benchmark dataset and naive prompting approach, the numbers and types of accessibility violations for each model are compared. This comparison helps to identify the differences in model behavior and analyze potential sources of bias or limitations in accessibility compliance. Through a qualitative analysis of the generated code, this question explores how various factors, such as training data biases, model instructions, and internal reasoning abilities, contribute to the variance in performance across different models.

RQ3: Do advanced prompt engineering and (post-)processing steps lead to more accessible MLLM-generated UI code? This question explores whether advanced prompting strategies can guide MLLMs in generating more accessible code. Particularly, it investigates the effectiveness of seven prompting strategies. Naive prompting as the baseline, zero-shot prompting with explicit accessibility instructions, few-shot prompting with examples of accessibility guidelines, chain-of-thought prompting to encourage step-by-step reasoning, and agentic prompting where multiple agents split the tasks of detecting, classifying, and solving violations. Lastly, two strategies use external automatic accessibility auditing tools to enhance their output: ReAct prompting, where the model iteratively critiques and improves its output based on violations found by accessibility tools, and color-aware prompting, which uses ReAct to critique its violations but further instructs the model with color contrast information and proposes potential solutions. Those strategies are evaluated based on the same benchmark dataset across the different MLLMs, and resulting violations are analyzed. The results show that prompting techniques are a controllable factor in improving accessibility violations, but also highlight potential side effects, such as cascading errors introduced during refinement steps. These findings serve as valuable insights for developers and researchers who aim to guide MLLMs towards more inclusive code generation.

RQ4: How consistent are the accessibility violations across different MLLMs on the same UI screenshot? To investigate the inter-model similarity and consistency of violations, this question compares the accessibility violations of the same UI screenshots across different MLLMs. By comparing the overlap and distribution of violations, it is possible to identify whether different models share the same weaknesses and blind spots. Therefore, this analysis uses the same WCAG 2.1 pipeline as in previous RQs, combined with the naive prompting strategy. The accessibility violations for each webpage and every model are combined into a

single vector whose dimensions count the violations per WCAG rule. The pairwise similarity between models and their corresponding webpages is then calculated using the cosine similarity of the violations vectors. While low similarity would reveal complementary failure modes that could be exploited by ensemble repair, high similarity would indicate systemic blind spots inherited from shared training data.

RQ5: Does data leakage influence the accessibility of MLLM-generated UI code? To rule out potential data leakage and influence on the accessibility of MLLM-generated code, this question compares the models' performance across two distinct datasets: the public benchmark dataset used in previous RQs, and a synthetic dataset composed of two subsets. The first subset is created through structural mutations. The second subset contains a fresh real-world dataset curated from open-source web projects, released after the knowledge cut-off of the affected MLLMs. By evaluating the code similarity and accessibility metrics across these datasets, this question tries to identify whether the performance is driven by memorization of training data or by true generalization. This analysis helps to reinforce findings of previous RQs and ensures that observed model behaviors reflect robust capabilities rather than overfitting to familiar data.

This thesis and its underlying empirical study demonstrate the potential and limitations of current MLLMs in generating accessible UI code from visual inputs. Motivated by these findings, this thesis discusses broader implications for designing and deploying generative models in web development. It intends to rethink accessibility as a primary design objective, rather than a post-hoc consideration. These perspectives offer possible directions for enhancing accessibility in the era of multimodal code generation.

1.3 Contributions

In summary, this thesis makes the following contributions:
Note that all experimental datasets, the code, and the results are made available on Github[1].

Accessibility Evaluation Pipeline: The first large scale accessibility study and evaluation pipeline for LLM-based Image-to-Code generation is proposed. This pipeline combines visual and structural fidelity with an automatic WCAG 2.1 conformity check. This comprehensive approach can be reused and extended in the future.

Realistic and Diverse Benchmark Dataset: This study uses a realistic dataset that contains 53 real-world webpage examples which have been collected from existing datasets and slightly mutated to minimize noise while preserving authenticity. It covers a wide spectrum of layouts, content areas and accessibility features, combining the screenshots and HTML/CSS code of each webpage. Additionally, a synthetic dataset is created to mitigate the risk of potential data leakage.

Model and Prompting Comparison: This study compares multiple MLLMs (both open-source and closed-source) across 7 prompting strategies, ranging from naive prompting to more advanced techniques. This thesis does not only quantify the results of the accessibility outcomes but also qualitatively analyzes violation patterns, and model-specific strengths and weaknesses.

[1] `https://github.com/marcolutz00/Image2Code`

Insights into Accessibility Limitations of MLLMs: This thesis provides insights into recurring WCAG 2.1 violations across different MLLMs and prompting strategies. It highlights systemic blind spots, which are likely caused by training data biases, and architectural constraints. Finally, a discussion and a possible guidance for researchers and developers will wrap up this thesis.

2 Methodology

In this section, the general methodology for evaluating the accessibility of HTML/CSS code is described. It includes a combination of automated auditing tools and manual verification. This hybrid approach identifies a wide range of accessibility issues, reconciles inconsistencies and guarantees reliable results. Figure 2.1 provides an overview of the evaluation pipeline, from data collection, prompt design to code generation, accessibility and similarity analysis, and post-processing. All evaluations are conducted following WCAG 2.1, the most widely adopted standard supported by the tools at the time of this thesis.

2.1 Automated Accessibility Evaluation

A wide range of automated accessibility checkers is available to detect violations in web content. According to former studies, automated testing can detect up to ∼60% of accessibility issues [9]. This makes them valuable for developers; however they can not completely replace manual testing or expert review. Regardless, in practice, a combination of various tools can help minimize the oversight of accessibility violations during the tests.
This study uses three widely adopted automated accessibility evaluation tools: *Axe-Core*, *Google Lighthouse Accessibility*, and *Pa11y*. Each tool offers unique detection mechanisms, coverage areas, and reporting formats.

- **Axe-Core (4.10.3) [33]:** is a rule-based engine developed by Deque Systems and commonly integrated into browser extensions and CI/CD pipelines. It provides detailed information and links each violation to its own rule set.
- **Google Lighthouse Accessibility (12.4.0) [18]:** embedded within Chromium-based browsers, performs accessibility audits alongside performance and SEO diagnostics.
- **Pa11y (8.0.0) [8]:** is a flexible command-line tool that uses HTML CodeSniffer as its engine and is especially useful for batch evaluation of static pages. During some tests on the benchmark dataset, it was found that Pa11y has strengths in areas where the other tools seem to fail (e.g., color contrast violations); therefore, it was decided to use it as the third complementary tool.

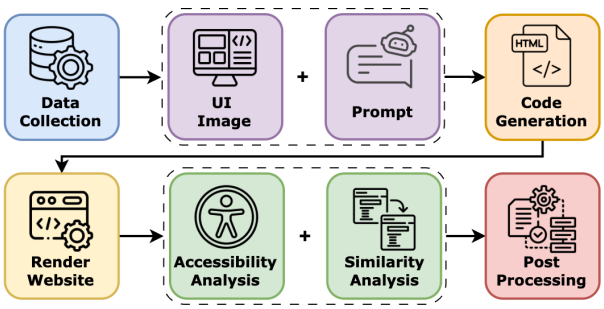


Figure 2.1: Overview of the evaluation pipeline, from data collection and prompt design to code generation, accessibility and similarity analysis, and post-processing.

All three tools were used to evaluate the generated HTML/CSS code. However, the outputs of each tool may differ. For example, Pa11y flagged a missing label for a form field under WCAG 2.1 Technique F68 (*"This form field should be labelled in some way."*), while Axe-core reported the same issue using its own rule IDs (e.g. *"label"*) and includes WCAG success-criterion tags such as *"4.1.2 Name, Role, Value"* in the rule metadata, but it does not cite WCAG techniques. This discrepancy reflects a common challenge in automated accessibility evaluation: the tools often differ in their interpretation, granularity, and labeling of the same underlying issues.
To address this inconsistency and improve cross-tool comparability, a unified taxonomy of accessibility issues is created. First, all detected violations are compiled to aggregate the outputs of the three tools. Based on the preserved metadata, such as rule IDs, WCAG mappings, descriptions, and affected HTML elements, the violations are grouped into functionally equivalent categories (e.g., missing labels, insufficient color contrast, missing alt-text). Next, similar rule definitions are combined into a single rule, leading to a consistent set of 40 accessibility categories, each mapped to one or more WCAG success criteria. For instance, issues such as *"H43.*"* (from pa11y) and *"empty-table-header"* (from axe-core) were all grouped under the category *"Table Headers"*. Similarly, contrast-related violations were merged under *"Color Contrast"*, according to WCAG 2.1 1.4.3 Contrast (Minimum), regardless of variation in wording across tools.

2.1.1 Manual Verification

As mentioned, automated tools can not completely replace manual testing. Certain accessibility guidelines, especially those involving contextual understanding or semantic meaning still require human judgment. Therefore, we support the evaluation with a structured manual review process. During the experiments, the generated HTML/CSS is audited manually by using the previously created accessibility issue taxonomy, especially identifying violations that may be overlooked by automated tools (e.g. improper headings, redundant links and semantically ambiguous structures). This manual layer of analysis helps to provide a more comprehensive and realistic assessment of the accessibility quality.

VIOLATIONS	TOOLS	RULE ID (TECHNIQUE ID)	DESCRIPTION
Color Contrast	Axe-Core/Lighthouse (*Pa11y*)	color-contrast (*G18, G145*)	Text/background contrast below WCAG AA threshold. (SC 1.4.3)
Landmark & Region	Axe-Core/Lighthouse (*Pa11y*)	landmark-one-main (–) landmark-unique (–) region (–) landmark-no-duplicate-main (–) landmark-main-is-top-level (–)	Landmarks missing, duplicated or incorrectly nested. (SC 1.3.1, 2.4.1)
Label Form Control	Axe-Core/Lighthouse (*Pa11y*)	label (*H44, H91, F68*)	Form control lacks an accessible label or label text. (SC 1.3.1, 4.1.2)
Headings	Axe-Core/Lighthouse (*Pa11y*)	page-has-heading-one (–) heading-order (*G141*) empty-heading (*H42, G130*)	Headings missing, empty or out of order. (SC 1.3.1, 2.4.6)
Distinguishable Links	Axe-Core/Lighthouse (*Pa11y*)	link-in-text-block (–)	Only color or a distinct styling distinguishes the link from the surrounding text. (SC 1.4.1)
Document Language	Axe-Core/Lighthouse (*Pa11y*)	html-has-lang (*H57*) html-lang-valid (*H57, H58*) html-xml-lang-mismatch (*H57, H58*) valid-lang (*H57, H58*)	Missing or conflicting page language declaration. (SC 3.1.1, 3.1.2)
Target Size	Axe-Core/Lighthouse (*Pa11y*)	target-size (–)	Interactive element's touch target is smaller than WCAG threshold. (SC 2.5.5)
Incomplete Links	Axe-Core/Lighthouse (*Pa11y*)	link-name (*H30, H91.A*)	Link has no perceivable name or content. (SC 2.4.4, 2.4.9, 4.1.2)
Image Alt-Text	Axe-Core/Lighthouse (*Pa11y*)	image-alt (*H36, H67*) input-image-alt (*H37*)	Image lacks alternative text. (SC 1.1.1)
Labels of Buttons	Axe-Core/Lighthouse (*Pa11y*)	input-button-name (*H91*) button-name (*H91*)	Button element has no accessible name. (SC 4.1.2)
Table Headers	Axe-Core/Lighthouse (*Pa11y*)	empty-table-header (*H43*) – (*H63*)	Table has missing, empty or incorrect referenced headers. (SC 1.3.1)
Document Title	Axe-Core/Lighthouse (*Pa11y*)	document-title (*H25*)	Page `<title>` element missing or empty. (SC 2.4.2)
Form Submit Button	Axe-Core/Lighthouse (*Pa11y*)	– (*H32*)	Form lacks a submit button. (SC 3.2.2)
Duplicate IDs	Axe-Core/Lighthouse (*Pa11y*)	– (*F77*)	Multiple elements share identical `id` attribute. (SC 4.1.1)
Incorrect List Structure	Axe-Core/Lighthouse (*Pa11y*)	list (*H48*) listitem (*H48*)	Incorrect list markup (ul/ol/li missing or nested wrongly). (SC 1.3.1)

Table 2.1: Taxonomy of the Top-15 automatically detected accessibility issues and their rule identifiers across the different tools. Axe-Core and Lighthouse report their violations based on the same rule set, while Pa11y uses WCAG techniques as reporting base which are shown in parentheses.

2.2 Dataset

2.2.1 Scope and Design

The main goal is to gather a diverse and high-quality dataset which consists of paired UI screenshots and HTML/CSS, and represents real-world webpages. The dataset should (1) include multiple domains and layouts, (2) contain annotated accessibility violations, and (3) have a reasonable size to be statistically relevant, but is also small enough to be analyzed manually. No publicly available dataset fulfills all requirements.

2.2.2 Construction

Two promising examples in the field of Image-to-Code are *Design2Code* [31] and *Webcode2m* [19]. Both represent real-world web interfaces and have been widely adopted in prior work on code generation and design understanding. Based on their dataset curation, both serve as a good base for this thesis.
To reduce redundancy, ensure layout diversity, and manage computational cost, a random sample of 28 instances from Design2Code and 25 instances from WebCode2M, resulting in a total of 53 UI–code pairs, is used for the study. This sampling strategy allows for a representative and feasible evaluation given the resource constraints of this thesis.

Content Distribution

Figure 2.2 summarizes the content distribution in our dataset. It contains a variety of domains, including blogs, business, and homepages, mirroring the diversity of real-world web content.

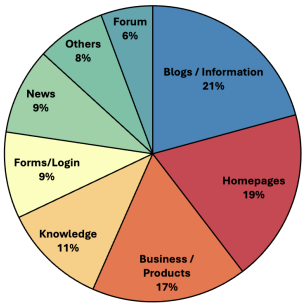


Figure 2.2: Distribution of Topics in Dataset

2.2.3 Dataset Alignment

Because Design2Code and Webcode2m use different strategies to purify their data, the datasets are harmonized before the analysis. The alignment contains (1) removing all external dependencies (e.g., images, audio, video, and external links) and substituting neutral placeholders (e.g., `src="placeholder.jpg"` for images or `href="#"` for links); (2) turning off executable content, such as scripts and other dynamic elements; (3) deleting non-visible content (e.g., advertisement-related tags or hidden elements) that are irrelevant in an image-to-code context and could otherwise possibly bias the accessibility metrics.

2.3 Benchmarks

The evaluation of the generated HTML/CSS is based on two complementary metrics: (1) Code Similarity and (2) Accessibility Metrics.

2.3.1 Code Similarity

Code similarity measures how closely the generated code matches the ground truth reference. This is crucial for assessing the layout fidelity and semantic correctness. Following the metric defined in [31], the evaluation is based on a combination of high-level visual similarity and low-level element matching, which captures both structural and stylistic alignment. The metric contains five components: text similarity (S_{text}), position similarity (S_{pos}), color difference (S_{color}), CLIP-based visual similarity (S_{clip}), and block size similarity (S_{block}). These scores collectively form the set $\mathcal{S} = \{S_{\text{text}}, S_{\text{pos}}, S_{\text{color}}, S_{\text{clip}}, S_{\text{block}}\}$. The final score is computed as a weighted sum of these components: $\mathit{CodeSim} = \sum_{i=1}^{5} w_i \cdot S_i$,, where $S_i \in \mathcal{S}$ denotes the i-th component score and w_i is its corresponding weight. Following the findings of previous work, we adopt a uniform weighting for all components in this study, setting $w_i = \frac{1}{5}$ for all i. The combination of those scores allows for a balanced view of the visual and structural similarity of the input and output.

2.3.2 Accessibility Metrics

Apart from counting violations, accessibility is evaluated along two further dimensions: the relative quantity of violations and their severity. The combination of both metrics allows us to understand whether LLMs can decrease not only the amount of accessibility violations but also their severity.

Inaccessibility Rate (IR): Following prior research [2]. It measures the percentage of DOM nodes with violations that exhibit at least one violation relative to the number of nodes prone to such violations:

$$\mathrm{IR} = \frac{N_{\text{violations}}}{N_{\text{total}}} \tag{2.1}$$

Impact-Weighted Inaccessibility Rate (IWIR): To capture the severity according to the WCAG impact levels, IWIR is introduced. Let v_i denote the number of violations at impact level $i \in \{\text{minor}, \text{moderate}, \text{serious}, \text{critical}\}$ and let $w_i \in \{1, 3, 6, 10\}$ be the corresponding weights. This scoring reflects the non-linear increase in impact for people with disabilities if a violation with a higher impact occurs. We normalize by the worst case for the observed counts:

$$\mathrm{IWIR} = \frac{\sum_{i=1}^{k} v_i \, w_i}{\sum_{i=1}^{k} v_i \, w_{\max}} \tag{2.2}$$

3 Empirical Study

Based on five research questions, a comprehensive empirical study was conducted to gain a better understanding of the capabilities and constraints of MLLMs in generating accessible UI code. The following presents the study setup and the quantitative and qualitative results.

3.1 RQ1: Do MLLMs generate accessible code from UI screenshots?

3.1.1 Experiment Setup

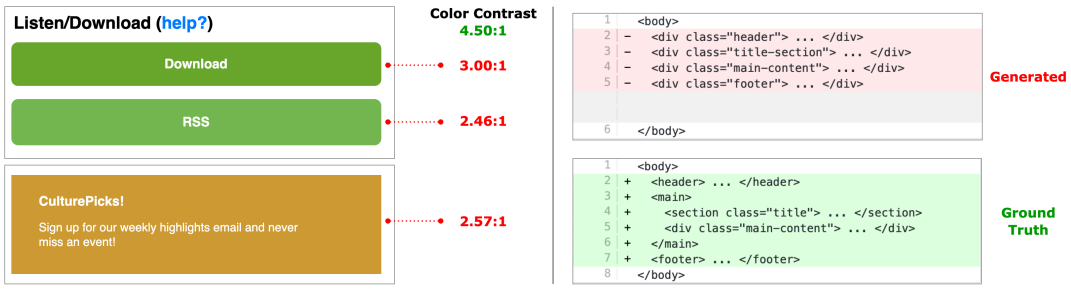
To evaluate the baseline capability of MLLMs in generating accessible code from visual UI input, two widely-used MLLMs are selected for this study: *GPT-4o* and *Gemini-2.0 Flash*. These models show strong performance in multimodal tasks, and their support for image inputs makes them suitable candidates for this study.
As described in section 2, the models are prompted with our dataset of UI screenshots and a corresponding task description. The task description includes a naive (plain) prompt, which instructs the model to generate HTML/CSS code based on the provided screenshot, without mentioning accessibility.

> **Naive Prompt**: Your task is to replicate the provided UI screenshot of a webpage pixel-perfectly using HTML/CSS.

This setup allows to observe whether MLLMs can naturally produce accessible code without direct instruction and acts as a benchmark for enhancement strategies. It is important to note that each experiment is conducted three times for each model, and the results below represent the average outcomes to account for stochastic fluctuations in the model's responses.

3.1.2 Results

Both models show a strong performance in reconstructing the given UI layouts with an average of 88.96% in code similarity for GPT and 87.12% for Gemini, as shown in Table 3.1. These results demonstrate that the models are capable of generating valid and visually similar HTML/CSS code from UI screenshots. While a more detailed evaluation of the visual fidelity is beyond the scope of this thesis, these results serve as a baseline for subsequent accessibility analysis. Table 3.1 presents the number of accessibility violations in UI code generated by GPT and Gemini. Despite achieving high layout and structural fidelity, both models produce a significant amount of accessibility issues. On average, GPT generates 13.75 violations per UI, while Gemini produces 15.45, or 12.4% more. The IR is 12.22% for GPT and 11.42% for Gemini, while IWIR is 47.10% and 47.70%, respectively. Although Gemini yields more total violations than GPT on average, an analysis of its DOM size reveals that it generates larger code snippets, which explains its lower IR because the violations are distributed over a larger number of nodes. However, the IWIR stays consistent across the two models, suggesting that the severity of the violations is similar. These findings demonstrate a clear gap between visual correctness and accessibility compliance and highlight that without clear and explicit instructions, MLLMs might not be able to follow accessibility principles in code generation. To understand the underlying reasons for these accessibility issues, a manual analysis of the violations observed in



a) Insufficient color contrast

b) Missing semantic landmark

Figure 3.1: Example of common accessibility violations.

the generated output has been conducted. This process makes it possible to identify recurring failure patterns and model limitations.

Most common violations

The detailed analysis reveals that two primary categories of accessibility issues are dominant in the generated code across both models. The first category is insufficient color contrast. For instance, color contrast violations are very common, causing 5.21 violations per UI for GPT and 6.89 for Gemini. The reason for this frequency can be attributed to the complexity of contrast checking, which requires the model to reason mathematically about RGB or hex values and their luminance for the human eye, as defined by WCAG. This mathematical requirement is not only challenging for MLLMs, but also human developers often rely on automated tools to ensure compliance in this area. The main difficulty lies in the fact that even slight deviations in color selection can lead to significant usability barriers for visually impaired users. As shown in Figure 3.1 (a), the generated code uses white text (#ffffff) on a green background (#67a52a and #72b650) or on a yellow background (#cc9933). This results in contrast ratios of 3.00, 2.46, and 2.57, significantly below the WCAG 2.1 minimum color contrast threshold of 4.5 for normal text. Even though this mathematical task is within the capabilities of MLLMs, it remains a common challenge across both models because of their limited visual-perceptual understanding and mathematical precision.
The second category is the lack or incorrect usage of semantic landmarks. To convert the visual structure of a webpage into a format that assistive technologies can use, semantic landmarks, such as `<main>`, `<nav>`, `<header>`, and `<footer>`, are essential. However, these landmarks are not visually displayed in the UI and have no specific representation, making it difficult for MLLMs, which rely only on screenshot input, to identify and generate them correctly. As shown in Figure 3.1 (b), rather than using semantic landmarks, the model substitutes them with generic `<div>` elements. As a result, assistive technologies interpret the webpage as a flat structure, without navigational and structural details, which are necessary for keyboard users or screen reader navigation. This can significantly hinder accessibility, especially for users who rely on semantic segmentation to jump between regions of a webpage. The issue basically stems from two factors: first, the lack of visible indicators, and second, the model's limited exposure to the underlying meaning of UI elements, which is the type of information that human designers often gain from accessibility training or work experience.

> ***Finding:*** Even though both GPT and Gemini achieve high layout fidelity, they consistently generate code with accessibility violations, especially in color contrast and missing landmarks. This indicates a fundamental gap between visual accuracy and accessibility compliance.

Violations	GPT	Gemini	Qwen	Llava
Color contrast	276	365	159	67
Landmark & Region	279	265	114	187
Label	102	105	21	14
Headings	26	18	20	17
Target Size	15	13	10	15
Distinguishable Links	20	23	2	0
Document Language	0	11	0	0
Table Headers	1	7	1	0
Incomplete Links	0	4	0	6
Image Alt-Text	4	5	0	1
Total Violations	729	819	329	307
Average Violations	13.75	15.45	6.21	5.79
IR	12.22%	11.42%	10.70%	12.14%
IW-IR	47.10%	47.70%	40.25%	40.40%
CodeSim	88.96%	87.12%	70.36%	50.50%

Table 3.1: Quantitative comparison of accessibility performance metrics across GPT-4o, Gemini-2.0 Flash, Qwen2.5-VL-7B, and Llava-7B.

3.2 RQ2: Do different MLLMs vary in their ability to generate accessible UI code?

3.2.1 Experiment Setup

To understand how the choice of MLLMs influences the generation of accessible HTML/CSS, the analysis is extended beyond the two closed-sourced models used in RQ1. This included two additional open-source models, *Qwen2.5-VL-7B* and *Llava-7B*, selected for their strong performance in multimodal tasks and wide adoption in the research community. The 7B parameter variations are used to capture substantial differences in model behavior and performance while ensuring the feasibility under constrained computational resources and reproducibility by other researchers. Similar to RQ1, all models are prompted with the same benchmark dataset and naive prompt, which instructs the models to generate HTML/CSS from UI screenshots without referencing accessibility. This setup allows a direct comparison across the models.

3.2.2 Results

Table 3.1 presents the average number of accessibility violations per UI, the IR, IWIR, and CodeSim for each model. Surprisingly, the open-source models Qwen and Llava have the fewest violations, averaging 6.21 and 5.79 violations per UI. Also, most of the other metrics show that these models perform better than the closed-source models: the average values for the IR are 10.70% for Qwen and 12.14% for Llava, while the IWIR is 40.25% and 40.40%, respectively. Since these results are significantly lower than the results of GPT and Gemini, a closer analysis was conducted, and three key factors influencing the accessibility results were identified: training data biases, instructional alignments, and code generation capabilities.

Training Data Biases.

Even if the internal training data of closed-source models is not publicly available, this qualitative analysis reveals noticeable differences in the models' behavior that can be attributed

```html
<!-- Similar to accessible reference (ground truth) -->

<img src="src/placeholder.jpg" alt="User image">
...
<label for="firstName">First Name</label>
<input type="text" id="firstName" name="firstName" required>
```

a) Accessibility-aware output by GPT-4o

```html
<body>
  <!-- Llava-style "flattened" markup -->

  <div class="header"> ... </div>
  <div class="navbar"> ... </div>

  <img src="src/placeholder.jpg">

  <div class="searchbar">
  <input type="text" placeholder="Search">
  </div>
</body>
```

b) Simplified template-like Llava markup

Figure 3.2: Comparison of semantically rich (left) and structurally shallow (right) HTML snippets

to training data biases.
The first visible difference is the tendency of Gemini to produce more violations related to color contrast, averaging 6.89 per UI for Gemini, compared to 5.21, 3.00, and 1.26 for GPT, Qwen, and Llava. This discrepancy seems to be closely related to the diversity and origin of the colors used in the generated code. For instance, Gemini shows a relatively limited color palette, consisting of only 41 unique colors across all UIs that cause violations. Over 40% of these violations stem from the use of the *primary blue* (#007bff) color, which is commonly used in the Bootstrap framework [5]. This strong dependency and bias towards a specific color suggests that Gemini's training data might have been disproportionately influenced by framework-centric codebases, such as those built with Bootstrap. As a result, the model shows a tendency to have stylistic preferences that do not align with accessibility. For instance, the Bootstrap primary blue is often used for elements like links, buttons, or headings, without taking the background color into account. Especially combined with light-colored layouts, this increases the risk of insufficient color contrast. In contrast, GPT's color contrast violations are caused by a set of 83 distinct colors. The majority of these colors (71.47%) are uniquely synthesized by the model and can not be linked to any framework or library. Even if this independent color selection does not prevent color contrast violations, it tends to more closely match the visual design of the input UI. This can lead to improved contrast ratios, given that the input UI is designed with accessibility in mind. While this interpretation is speculative because of the black-box nature of the models, the observed patterns suggest a strong influence of the training data on the models' accessibility outcomes.

Instructional Alignments.

The second key factor appears to be the instructional alignment and the behavior tuning of the models during training. GPT frequently generates code with accessibility-conscious elements (e.g., alt-text for images or proper form labels), even when prompted without explicit instructions. As shown in Figure 3.2 (a), GPT tends to produce descriptive alt-text and labels by default.
While the reasons remain speculative, a plausible explanation could be GPT's accessibility-oriented use cases and alignment practices. For example, GPT has been used in real-world assistive technologies like *Be My Eyes* [3]. This application helps to interpret visual information into textual descriptions, which is crucial for visually impaired users. This practical integration into a real-world scenario might have influenced the model's fine-tuning or evaluation objectives. These improved accessibility considerations and sensitivity could be a result of accessibility-focused tasks during the training process of the model. This characteristic is less noticeable in other MLLMs.

Code Generation Capabilities.

Despite generating code with fewer violations, the open-source models Qwen and Llava do not necessarily produce accessible webpages. The detailed analysis reveals that these models tend to generate simplistic and semantically shallow code snippets. They often generate flat layouts or generic tags, such as `<div>` or `<span>`, combined with little nesting, styling, or ARIA annotations as default. This behavior is illustrated in Figure 3.2 (b), where Llava reduces a multi-component UI layout into a linear sequence of `<div>` elements, following a common template. This approach leads to fewer accessibility violations because it lacks problematic elements and therefore reduces the surface for potential issues. However, this results in structurally incomplete and visually incorrect code, which can be seen in significant lower code similarity scores of 70.36% for Qwen and 50.50% for Llava, compared to 88.96% for GPT and 87.12% for Gemini. This indicates that the smaller, open-source models fail to capture layout fidelity and semantic depth. This finding highlights the necessity to balance accessibility evaluation with structural quality, because the absence of content can artificially reduce the violation metrics.

Finding: Due to variations in alignment goals (e.g., GPT's accessibility-aware behavior) and training data composition (e.g., Gemini's dependence on framework-derived styles), accessibility violations differ significantly amongst MLLMs. Open-source models, such as Qwen and Llava, tend to produce simplified, under-specified code, which is the main reason why they report fewer violations.

3.3 RQ3: Does advanced prompt engineering lead to more accessible MLLM-generated UI code?

3.3.1 Experiment Setup

While the prior RQs explored the inherent capabilities of MLLMs with naive prompting, this RQ investigated the prompting strategies' impact on the accessibility of the generated code. Prompt engineering has shown to be a powerful technique in prior work to improve and influence the behavior of LLMs, especially in structured tasks. Therefore, this study takes 7 different prompting strategies into account, ranging from naive to externally supported techniques. Note that the prompts follow the established best practices of prior research [32, 40] and that all prompting techniques are tested on GPT and Gemini by using the same benchmark dataset as in previous RQs.

Naive Prompting (Baseline): Without explicit accessibility instructions, the model is instructed to generate HTML/CSS code based on a screenshot of a webpage.

Zero-Shot Prompting: The MLLMs are explicitly instructed to generate accessible code by adding the instruction *"comply with WCAG 2.1 standards"*, as shown in Figure 3.3. This explores the model's ability to generalize accessibility guidelines only based on an instruction and without examples.

Few-Shot Prompting: This strategy resembles the zero-shot prompt, but is enriched with several examples of WCAG 2.1 guidelines, combined with a description of the task, an incorrect code snippet, and a corresponding correct code snippet. The examples are sourced from the ARIA Authoring Practices Guide (by W3C) [23] and Accessible Components by Gov.uk Design

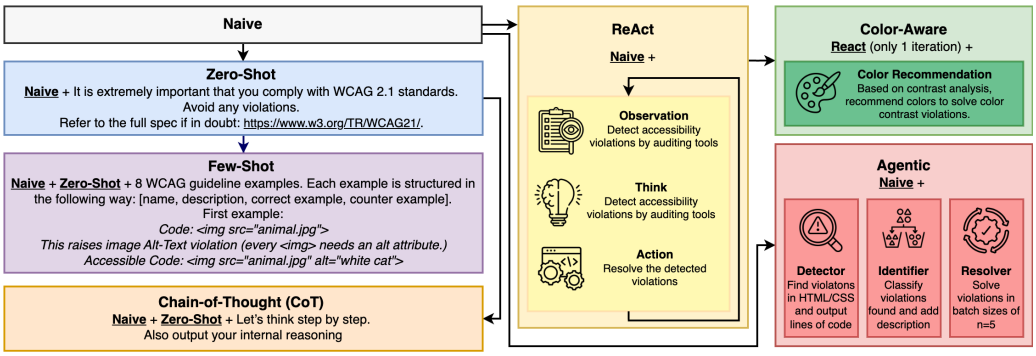


Figure 3.3: Overview of the advanced prompting strategies.

System [30], which are known for their accessibility best practices. Eight examples, which differ in their content (e.g., image alt-text, form labels, landmarks), are manually collected. They serve as an additional context for the model to lead the models towards a more accessible code generation.

Chain-of-Thought Prompting: As shown in Fig. 3.3, the prompt encourages the model to think step-by-step. The objective is to guide the model to describe its considerations in a structured plan and finally output the code. This helps to show intermediate reasoning steps that possibly highlight accessibility-aware thinking.

Agentic Prompting: As prior research [39] has shown, the use of multi-agent systems, where each agent has a specific task, can lead to better results in complex tasks. Therefore, this strategy splits the task of creating accessible code into three agents: The *Detector* agent is designed to detect accessibility violations in a generated code, instructed with the naive prompt. It outputs a list of snippets with violations, including their location in the code. The second agent, the *Identifier*, is responsible for classifying the violations into their respective WCAG guidelines (e.g., color contrast, landmarks, etc.) and enrichich the each violation with its severity level. The third agent, the *Resolver*, is instructed to resolve the list of violations in the code. In order to prevent cascading errors, this agent is instructed to solve batches of violations, having a size of $n = 5$. The goal of this strategy is to evaluate whether decomposing the task into smaller and more manageable subtasks can improve the overall accessibility.

React Prompting: This strategy first instructs the model to generate HTML/CSS code, according to the naive prompt, and then to critique its accessibility violations detected by the automated tools. As shown in Fig. 3.3, the model is then asked to revise all violations found, prioritize those with the highest severity, and output the revised code. This multi-step approach evaluates the capability of the model to self-correct and iteratively apply accessibility standards. Each self-refinement loop runs for three iterations or until no further violations are detected.

Color Aware Prompting: As previous RQs have shown, color contrast violations are a common issue in MLLM-generated code. Due to their mathematical nature, they can be difficult to solve. Therefore, this strategy extends the React prompting by adding a color-aware step. It uses a pre-processing step that extracts the color values from the screenshot, analyzes the color contrast violations found, and provides the model with a possible solution for each violation. Similar to the React prompting, the model is then instructed to output the revised code. By supplying the model with extracted color values and recommended replacements that satisfy WCAG thresholds, the objective is to decrease color-contrast accessibility violations. The model

is directed to generate code that maintains visual consistency while utilizing compliant color pairs by supplying this analysis and asking for a revision. Note that the refinement loop runs only for one iteration.

3.3.2 Results

Table 3.2 presents the average number of accessibility violations (AV), total violations (TV), code similarity (CodeSim), inaccessibility rate (IR), and impact weighted inaccessibility rate (IWIR). The results demonstrate that advanced prompting techniques can significantly reduce the number of accessibility issues in the generated code. For instance, when using the most advanced prompt (React), 95.75% of the violations for GPT and 82.07% for Gemini are resolved, compared to their naive counterpart. A similar trend can be observed for the IR and IWIR. While the IR is reduced to 0.44% for GPT and 1.36% for Gemini using the React prompt, the IWIR is reduced to 13.56% and 21.48%, respectively. However, the effectiveness of each prompting strategy varies across the models, and some strategies are more suited for certain models than others.

Technique	**GPT**					**Gemini**				
	AV	TV	CodeSim	IR	IWIR	AV	TV	CodeSim	IR	IWIR
Naive	13.75	729	**88.96%**	12.22%	47.10%	15.45	819	**87.12%**	11.42%	47.70%
Zero-Shot	12.33	653	87.79%	10.02%	47.35%	14.25	755	86.85%	10.33%	47.74%
Few-Shot	9.49	503	87.29%	7.96%	44.64%	15.74	834	86.95%	10.93%	47.91%
Chain-of-Thought	10.04	532	87.91%	8.61%	44.53%	14.28	757	86.80%	11.07%	48.31%
Agentic	11.34	598	86.12%	8.86%	41.80%	13.16	693	85.07%	8.36%	47.00%
ReAct	**0.68**	**36**	86.94%	**0.44%**	**13.56%**	2.77	147	86.16%	**1.36%**	**21.48%**
Color-Aware	3.35	177	86.42%	2.35%	33.68%	**2.58**	**137**	85.59%	1.74%	34.93%

Table 3.2: Performance for advanced prompting techniques, including Average Violations (AV), Total Violations (TV), Code Similarity (CodeSim), Inaccessibility Rate (IR), and Impact Weighted Inaccessibility Rate (IWIR).

Impact of Zero-Shot Prompting

All models show a moderate, but measurable improvement in accessibility violations when a simple instruction about accessibility is added to the prompt. For instance, GPT reduces the average amount of violations by 10.33% and Gemini by 7.76% compared to the naive prompt. However, while the IR shows a similar trend with a considerable reduction for both models, the IWIR remains almost unchanged, even showing a slight increase. This suggests that even if the models are capable of activating their accessibility awareness, even without explicit examples, they still struggle to solve the most severe violations.
A deeper analysis of the violation types and their distribution reveals that some limitations for the models remain. For example, both models often use accessibility-related attributes (e.g., label, alt, role, aria-*) but misuse them and fail to apply them in a consistent manner. Figure 3.4 demonstrates a representative example. While Gemini generates an e-mail subscription form field, it fails to provide any accessible name, e.g, neither a `<label>` nor an `aria-label` attribute. This indicates that zero-shot prompting triggers awareness, it does not necessarily resolve accessibility gaps and inconsistencies.

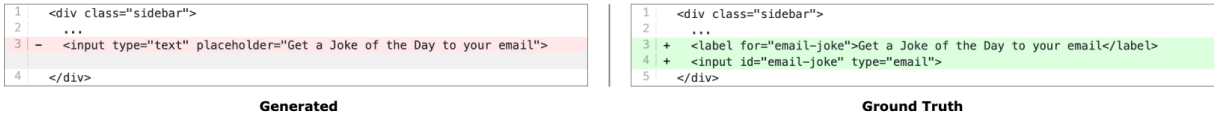


Figure 3.4: Accessibility gap in zero-shot prompting: omission of an accessible label in the generated code compared to the correctly labeled ground truth.

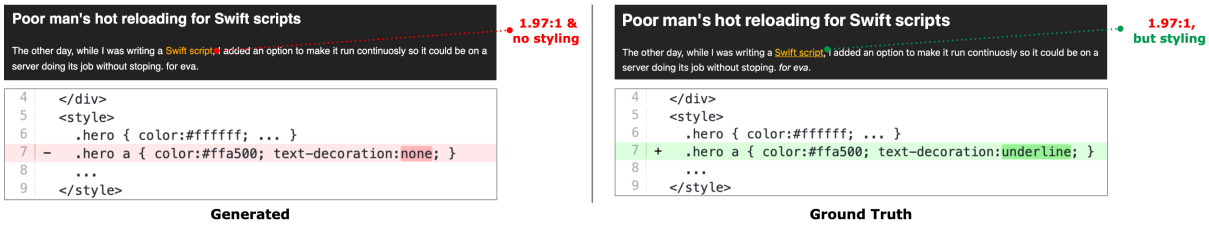


Figure 3.5: Example of a Distinguishable Links violation in Gemini's few-shot output. The generated code does not include the underline and relies solely on a low-contrast orange (1.97:1), whereas the ground truth includes the underline to ensure link distinction despite insufficient contrast.

Impact of Few-Shot Prompting

The few-shot prompting technique results in a 30.98% improvement in accessibility performance for GPT, with the average number of violations dropping from 13.75 to 9.49, while the IR decreases from 12.22% to 7.96%. By observing eight high-quality, concrete examples, the model is capable of copying the underlying patterns and structure. For instance, when provided with an example of correct heading structures, such as `<h1>`, GPT is able to incorporate this pattern into its generation, reducing the average number of heading violations from 26 to 15. Furthermore, the examples presented reflect only important accessibility guidelines, which significantly impact users and therefore also a have high severity. This is reflected in the IWIR, which drops from 47.10% to 44.64%. This demonstrates the importance of concrete examples in guiding the model, especially for concepts like accessibility, which may be underrepresented in the training data.

However, surprisingly, Gemini does not show similar improvements. It performs slightly worse with an increase of 1.88% in the average number of violations. Based on the qualitative analysis, this effect can be attributed to two primary factors. First, Gemini exhibits a 56.52% increase in violations related to the *Distinguishable Links* category. Due to its exposure to link-related few-shot examples, it tends to generate links more frequently, but often without sufficient contrast or styling, violating the WCAG requirements. This phenomenon may be directly linked to the color bias in the training data, which has been observed in RQ2. Gemini tends to use generic, default colors for links, such as the primary blue from the Bootstrap framework, which used to be the primary color for links in this framework. Since it also fails to apply distinct stylings for the links, this combination leads to a lack of distinction between links and the surrounding text. As Figure 3.5 illustrates, Gemini generates links with insufficient contrast and no distinct styling.

The second violation category *Landmark & Region* increases by 12.83%, due to Gemini's tendency to misuse the semantic elements, such as `<main>` or `<nav>`. This can likely be attributed to a misinterpretation of the few-shot examples, which might be replicated directly without a deeper semantic understanding of how those elements contribute to accessibility.

Impact of Chain-of-Thought Prompting

Chain-of-Thought (CoT) prompting results in notable improvements over the naive prompt, i.e., averaging 10.04 vs 13.75 violations per UI for GPT and 14.28 vs 15.45 for Gemini. Due to the reasoning instruction before generating code, the models are able to reflect possible accessibility constraints often missed in direct code generation. Interestingly, GPT benefits significantly more than Gemini, with a relative improvement of 26.98% vs 7.57%. This is probably because GPT constantly outlines an organized structure to reasoning by recognizing particular accessibility issues (such as contrast ratios and alt text) and integrating them into the code that is generated. In contrast, Gemini's CoT reasoning is less consistent and frequently omits key steps or fails to use the reasoning effectively during code generation.

Impact of Agentic Prompting

Splitting the task into three agents only leads to moderate improvements, as GPT reduces its violations by 17.53% and Gemini by 14.82% compared to the naive baseline. Similar to the prior strategies, the IR follows the same trend, dropping from 12.22% to 8.86% for GPT (27.50%) and from 11.42% to 8.36% for Gemini (26.80%), but the IWIR decreases much less, with 11.25% for GPT and only 1.47% for Gemini.
The qualitative analysis reveals two main reasons for these results. First, the Detector agent misses some violations in the generated code and occasionally even reports false positives. As a result, the Resolver agent either does not resolve all violations within the code or changes already correct markup, which leads to new violations. Second, the Resolver resolves the violations in batch sizes. This can prevent the Resolver from understanding layout dependencies, leading to small but significant changes that decrease the layout fidelity, which results in the lowest code similarity of all prompting techniques (86.12% for GPT and 85.07% for Gemini). In general, while the agentic prompting reduces the load per step of the models, it introduces additional noise that propagates through the agents.

Impact of React Prompting

The experiment shows that both GPT and Gemini significantly improve their accessibility performance when using the React prompting strategy. Across three iterative refinement cycles, GPT achieves reductions of 82.33%, 92.07%, and 95.05% in accessibility violations compared to the naive prompt, while Gemini improved by 62.14%, 77.67%, and 82.07%, respectively. Similar improvements can be observed for the IR and IWIR. After three iterations, only 0.44% of nodes in GPT's code contain accessibility violations, while Gemini remains with 1.36%. Interestingly, not only the number of violations but also their severities are reduced. For instance, the IWIR drops from 47.10% to 13.56% for GPT and from 47.70% to 21.48% for Gemini. This indicates that the model is not only capable of resolving violations, but also prioritizes the violations with severity *serious* or *critical*. After three refinement iterations, neither GPT nor Gemini produces any violations with severity *critical*, and only few with *serious*.
The results suggest that even if models do not inherently obey all accessibility guidelines, they can be guided towards more accessible code by incorporating external observations of the automated tools. GPT consistently outperforms Gemini across all iterations, possibly due to its stronger alignment with Reinforcement Learning from Human Feedback (RLHF), which equips it with a better capacity to self-correct and obey instructions based on user-oriented feedback.
A notable observation from the React prompting is that the two main categories of accessibility violations, color contrast and landmarks, are significantly reduced. While landmark violations are reduced by 98.21% for GPT and 95.85% for Gemini, color contrast violations are reduced

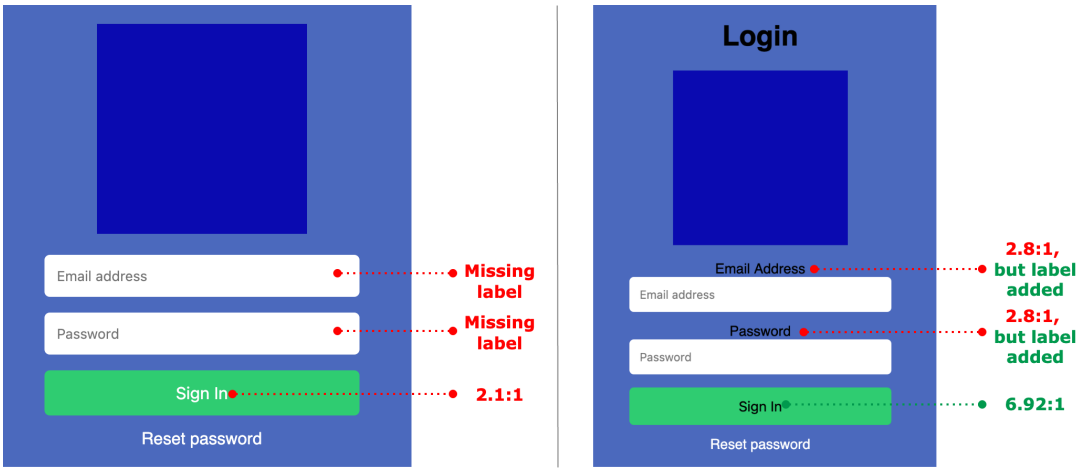


Figure 3.6: Login screen before (left) and after (right) the first refinement iteration. The model successfully solves the missing label violation, but introduces a new issue with the color contrast of the label.

by 90.58% and 64.93%, respectively. These results reflect a growing capacity of MLLMs to add feedback into their stylistic choices, which leads to more accessible and visually consistent webpages.
However, the React prompting also has its limitations. Even though both models decrease the dependency on framework-based colors, the results still vary across the models. For instance, especially during the first refinement iterations, Gemini still heavily relies on framework-based color palettes, especially Bootstrap's primary blue, which leads to an increase in those violations of 23% with this exact color, even though overall the amount of color contrast violations is reduced by 22%. This finding suggests that the model might not be able to synthesize its own new colors, but rather tries to substitute problematic colors with known ones from its training data. Furthermore, some attempts to fix particular violations can occasionally lead to new problems elsewhere in the code. As Figure 3.6 suggests, the model successfully solves certain violations (for example, adds a `<label>` to the form element), but at the same time introduces a new issue with the color contrast of the label. These cascading errors may arise from a limited understanding of the code's underlying structure and dependencies, which highlights a potential limitation of current MLLMs.

Impact of Color-Aware Prompting

Enhancing the React prompting technique with pre-computed and WCAG-compliant colors has a different impact on both models. While Gemini struggles to invent new colors, it benefits the most with its average number of violations dropping by 83.30%, compared to the naive prompt, and is even able to reduce its number by 6.86%, compared to the React prompting. While GPT also benefits from this strategy, by reducing its violations by 75.64%, it does not match the improvements of the React prompting strategy.
Two main reasons can be attributed to this finding. First, GPT is already able to synthesize new colors. Providing new externally computed colors occasionally leads to conflicts elsewhere in the code, as the model tries to incorporate these colors. Second, the bounding-box detection of *Design2Code* occasionally groups elements together that are not visually connected in the UI. This can lead to situations where the model applies external colors for an entire region, rather than a specific node. Overall, the results suggest that adding explicit WCAG-compliant colors can significantly lower contrast violations. However, the underlying model's design capabilities

and accuracy determine how beneficial this strategy is.

Finding: Prompting strategies significantly influence the accessibility of MLLM-generated UI code. While the models benefit from explicit guidance, advanced techniques like React prompting yield the most substantial improvements. However, each strategy may also introduce model-specific limitations and, in some cases, create new issues. These findings underscore the importance of aligning prompting strategies with model capabilities to enhance accessibility outcomes.

3.4 RQ4: How consistent are the accessibility violations across different MLLMs on the same UI screenshot?

3.4.1 Experiment Setup

The objective of this RQ is to find out whether different MLLMs cause similar or complementary accessibility violations when asked to generate HTML/CSS from the same UI screenshot. Consistent error patterns would indicate that systematic limitations, caused by similar training data, exist. In contrast, complementary violations would motivate ensemble or shared approaches between the models to improve the overall accessibility of the generated code. To investigate this, combined with the naive prompting technique, the benchmark dataset is used as input for GPT and Gemini. For every tuple of screenshot and model, the list of accessibility violations is extracted, sorted and, represented by a violation vector, which counts the number of issues per WCAG class.

Violation Vectors: Each violation vector is a numerical representation of the accessibility violations found in the generated HTML/CSS. Given a model-screenshot pair, each dimension corresponds to a specific violation class, and its value represents how many violations of that class were found. This representation makes comparing error patterns across different models and screenshots easier. The violation vector is defined as follows:

$$\begin{pmatrix} \text{Issue}_1: & x_1 \\ \text{Issue}_2: & x_2 \\ \vdots & \vdots \\ \text{Issue}_k: & x_k \end{pmatrix} \xmapsto{\text{to vector}} \mathbf{v} := \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{pmatrix} \in \mathbb{R}^k.$$

To compare the similarity of the violation vectors, the cosine similarity is used, a common metric in other research fields.

Cosine Similarity Given two vectors with $\mathbf{x}, \mathbf{y} \in \mathbb{R}^k$, the cosine similarity is defined as:

$$\text{cos\_sim}(\mathbf{v}_1, \mathbf{v}_2) = \frac{\mathbf{v}_1^\top \mathbf{v}_2}{\|\mathbf{v}_1\| \, \|\mathbf{v}_2\|} \quad \in [0, 1].$$

3.4.2 Results

Figure 3.7 illustrates the cosine similarity matrix plotted as a heatmap, where each cell represents the similarity between the violation vectors of different model runs. Overall, most cells are light-colored, indicating a high cosine similarity between the runs of the models, suggesting that GPT and Gemini produce similar violation patterns.
This is especially true for the upper part of the heatmap, representing the dataset entries of the Design2Code dataset (examples 1-28). Here, both models show a high degree of consistency

and similarity across each other and also across the individual runs. The lower part of the heatmap, representing the dataset entries from the dataset Webcode2m, displays a more diverse pattern, with a mix of lighter and darker cells. Nonetheless, even in this section, the majority of the cells are light-colored, showing that the models are still largely consistent in their violation patterns.
A manual inspection of the darker cells reveals that a great number of lower similarity scores stems from two main reasons: 1) In some runs, a model is capable of generating code with only few violations, leading to a violation vector with many zeros or small values. Because the cosine similarity is highly sensitive to small magnitudes, even another run with a comparable low number of violations can result in a lower cosine similarity score. This effect is especially true for Webcode2m examples, where the models generally produce fewer violations. 2) The second reason for lower similarity scores stems from possible recurring errors in the code generation. For instance, if one run uses a non-WCAG-compliant color palette, the resulting number of color contrast violations might be considerably higher than in other runs with more compliant colors. This difference in a single category can significantly reduce the cosine similarity, even if other categories show comparable patterns.
Overall, the results suggest that the models generate code snippets with very similar numbers, categories, and distributions of accessibility violations. The high similarity score indicates that the models share systemic limitations and biases, which are likely to be inherited from the underlying training data.

Finding: Accessibility violations are highly consistent across GPT and Gemini, with an average cosine similarity of 0.719. This suggests that the models inherit the same systemic blind spots from web-scale training data.

3.5 RQ5: Does data leakage affect the accessibility of MLLM-generated UI code?

3.5.1 Experiment Setup

A concern that is often raised in the context of MLLMs is the possibility of data leakage, where the models have already been exposed to the test data during pretraining, thus leading to inflated performance. This RQ aims to investigate whether such leakage has an impact on the accessibility of the generated code.
To investigate this, two distinct datasets have been created to compare model performance. 1) **Benchmark Dataset**, which has been used and described in the previous RQs. 2) **Synthetic Dataset**, which consists of two distinct subsets. The first subset is created using layout mutation and hybrid synthesis techniques. Therefore, heuristic transformation rules have been created to mutate layout structures, elements arrangements, component colors, and text content. In a second step, some layouts and components are then interchanged across the UIs, which leads to new compositions. It contains new UI compositions and code that are not likely to have exact matches in the pretraining data. An example of this synthetic dataset is shown in Fig. 3.8. As a result, 10 webpages are synthesized, which are rendered into screenshots and paired with their corresponding mutated HTML/CSS code. The second subset is created to complement more up-to-date and real-world examples in the synthetic dataset. This subset is curated by selecting two open-source web projects [24, 16] that are created after the official knowledge cutoff dates of the MLLMs. These projects contain new samples that are unlikely to

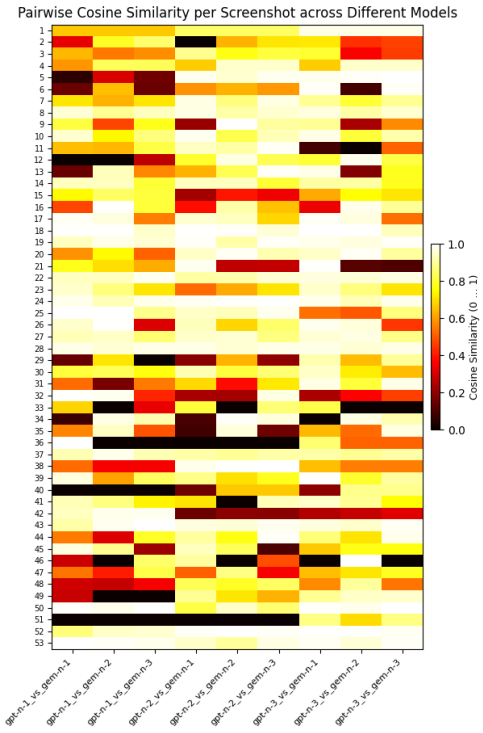


Figure 3.7: Cosine similarity heatmap between the 3 runs of GPT and Gemini on the same UI screenshot. Each cell represents the cosine similarity of the accessibility violation vectors generated by the models. Light = high similarity, dark = low similarity.

have been seen during pretraining. However, web code from open repositories often includes dynamically generated content, external dependencies, and redundant content, which can bias this accessibility-focused analysis negatively. Therefore, the collected code is manually cleaned according to the previous steps, described in Section 2.2. Through this process, 10 new webpages are collected, and their corresponding HTML/CSS together with the rendered UI screenshots are added to the subset.
The two subsets are then combined into a single synthetic dataset, which is used to compare the models' performance against the benchmark dataset.
All experiments use the same naive prompting setup and are conducted on GPT and Gemini. By comparing the code similarity and distribution of accessibility violations across the different datasets, it can be evaluated whether the performance variations suggest memorization or data leakage effects.

Dataset	**GPT**			**Gemini**		
	CodeSim	IR	IWIR	CodeSim	IR	IWIR
Benchmark	88.96%	12.22%	47.10%	87.12%	11.42%	47.70%
Synthetic	89.17%	11.64%	49.47%	88.01%	12.06%	50.30%

Table 3.3: Code similarity (CodeSim) and accessibility metrics (IR, IWIR) on the Benchmark and Synthetic datasets.

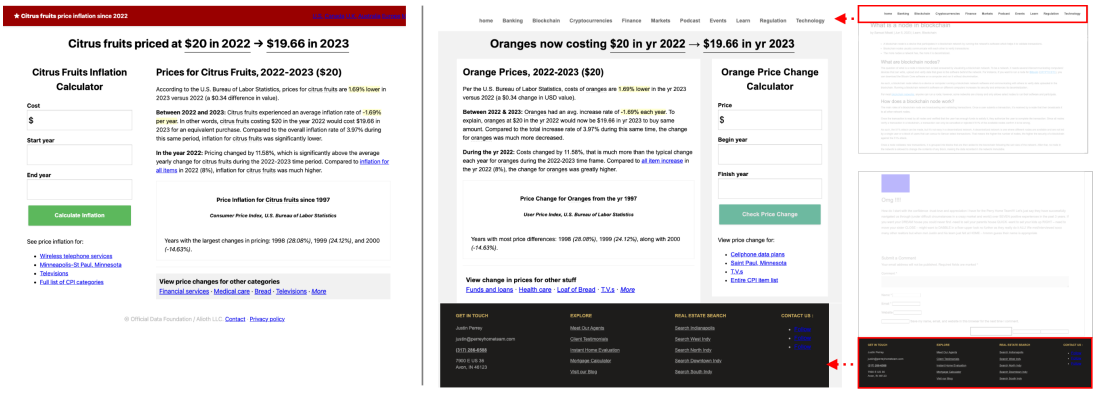


Figure 3.8: Original web page (left), a heuristically mutated version (middle) with altered color, layout and text, and a cross-site hybrid composition combining sections from multiple pages (right).

3.5.2 Results

Table 3.3 shows the results across all three datasets. Overall, the code similarity scores are relatively consistent across the datasets, even showing slightly higher values for the synthetic dataset. While GPT achieves a code similarity of 88.96% for the benchmark dataset vs 89.17% for the synthetic dataset, Gemini achieves 87.12% vs 88.01%, respectively. This suggests that the models are not merely memorizing and reproducing the training data, but rather are able to generalize and synthesize new code. The IR changes by 0.5-0.6 percentage points (GPT slightly lower, Gemini slightly higher), which is considered within the variance of the models shown in RQ3. This indicates that the number of DOM nodes with at least one accessibility violation is largely unaffected by the underlying dataset. Lastly, both models show a modest increase in the IWIR, with GPT increasing from 47.10% to 49.47% and Gemini from 47.70% to 50.30%. Manual inspection reveals that the synthetic dataset, especially the 10 mutated UIs, contains more composite sections, which tend to trigger slightly more color contrast violations, and therefore lead to a higher IWIR.
These findings demonstrate comparable performances in both code fidelity and accessibility metrics across the different datasets. This suggests that the results shown in previous RQs are not artificially inflated by data leakage, but rather reflect the overall generalization capabilities of the models. Therefore, it can be concluded that data leakage is unlikely to exist or affect the evaluation process of MLLMs in this thesis.

> ***Finding:*** The consistent performance of MLLMs across the benchmark and synthetic datasets suggests that data leakage is minimal and the models' behavior observed in earlier experiments is not a result of memorization or pretraining exposure.

4 Discussion

This section discusses the findings of this thesis and explores possible implications for the future of accessibility-aware code generation.

4.1 Rethinking Accessibility as a First-Class Objective in Code Generation

Even though state-of-the-art MLLMs have shown impressive capabilities in generating UI images into syntactically valid HTML/CSS code, accessibility remains a significant challenge. When prompted natively, even the most advanced models, GPT-4o and Gemini-2.0 flash, show only little awareness of accessibility best practices. These models often generate code that is similar to the original UI image but consistently fails to meet the accessibility standards set by the WCAG. This highlights the necessity of a paradigm shift in the approach of code generation. In practice, the conclusions of this thesis often reveal that improving the overall model quality is not sufficient to guarantee accessibility. Functional and visual correctness of the generated code can be orthogonal, or even in conflict with accessibility. A layout that renders correctly, might still not be navigable by keyboard users or usable for screen reader users.

Therefore, these findings highlight the need to rethink accessibility as a first-class objective in model development, prompting, and evaluation. This shift means more than just incorporating accessibility checks as a post-processing step, but rather integrating it proactively into the design principles of the models. For instance, this would include that accessibility metrics should be reported alongside traditional code quality metrics, especially in domains where accessibility is crucial, such as web development.

To get insights from developers working on a daily basis with LLMs, a small survey and discussions [14, 27] has been conducted in a couple of developer communities. The feedback from these discussions and some private interviews highlights how LLMs are perceived in practice: *"LLMs can point you in the right direction, but they struggle with specific fixes"*. Others cautioned that *"over-reliance without fact-checking will ultimately worsen the experience for the user"*. This feedback underlines an important point: while LLMs can assist and accelerate the development process, they can not yet be trusted to ensure accessibility without human oversight. This highlights the necessity for accessibility to be an optimization goal in the entire code generation process.

4.2 From UI Description to Code: An Even Greater Accessibility Challenge

While the scope of this thesis focused on the evaluation of generated code from UI screenshots, recent advancements could shift the focus from *image-based generation* to *description-based generation*. In this context, the model is prompted via natural language or structured descriptions and generates code directly from these inputs. However, this shift introduces even greater challenges and a greater risk of omission of accessibility features.

Unlike with screenshots, where the model can visually get information about layout and stylistic details, text descriptions may not mention accessibility requirements at all. The responsibility of including accessibility information falls entirely on the developer, who must make sure that the description contains all necessary details. However, this includes the assumption that the developer is aware of the accessibility requirements, which is often not the case. For instance, a developer might write *"Create a login form with username and password fields"*, without mentioning that specific labels, keyboard focus orders, or ARIA roles are required. Additionally, accessibility-related information, such as the choice of colors, alt-texts for images, or keyboard navigation details, can be dependent on the context and subjective for each individual developer. Understanding these details from language alone is a significant challenge for LLMs. For example, a model might not be capable of determining which elements on a webpage should be reachable via keyboard or if the color contrast is sufficient for the user.
Therefore, the shift from image-based to description-based code generation poses even more challenges for accessibility and demonstrates the need for explicit alignment and prompt design to ensure that accessibility considerations are adequately addressed.

5 Threats to Validity

The chapter introduces possible threats to the validity of this thesis and demonstrates how these threats are mitigated.
First, the representativeness of the UI dataset poses a potential threat to the generalizability of the findings. In order to reduce this threat, the data entries are collected from two publicly available datasets (Design2Code and Webcode2m). These two datasets are manually inspected and found to contain a diverse range of UI designs, of which 53 examples are selected for the study. Those examples cover a wide range of UI elements, design, and content types. Additionally, a synthetic dataset, consisting of 20 entries, is created to further increase the diversity and rule out the risk of potential data leakage.
Second, the choice of models poses another potential threat to this thesis. To address this, the study includes four widely-used and state-of-the-art MLLMs, such as GPT-4o, Gemini-2.0 flash, Qwen2.5-VL-7B, and Llava-7B. The thesis has intentionally not included task-optimized or fine-tuned models to reflect realistic usage scenarios. Apart from that, possible resource constraints are also considered. The thesis evaluates the open-source models at the scale of 7B parameters, rather than using their large variants (e.g., Qwen-2.5-VL-72B), which might show a stronger performance, but are less accessible to most users. This selection of models captures the diversity of MLLMs at the current state, but also acknowledges that further model families and variants could enhance the representativeness of the findings.
Third, the models' prompting strategies pose another potential threat to the validity of the findings. To reduce the risk of prompting bias, the thesis has evaluated the performance of the models with a set of 7 different prompting techniques, ranging from a naive prompt without any additional accessibility instructions to more advanced prompts that include external accessibility checkers and guidelines. While there are many more techniques that have shown promising results in the literature, such as *prompt tuning* or *model context protocol* (MCP), the resource constraints of this thesis limit the number of techniques that could be evaluated. The prompt design used in this study, however, was intended to be representative of general use and aligned with best practices of prior research.
Lastly, another threat to validity is the probabilistic and non-deterministic nature of MLLMs, which can lead to variations in the observed results, even for the same input. To mitigate this threat, the study has evaluated the performance of each model based on three different runs and reported the aggregated, averaged results. While this approach helps to reduce the number of possible outliers, it does not fully eliminate them. Overall, the observations in this thesis showed variations between the runs, but the overall trends and findings remained consistent. Nevertheless, a larger number of runs could further improve the robustness of the findings.

6 Related Work

The related work in this chapter is structured into two main areas: 1) studies in web accessibility and 2) studies of MLLMs for UI code generation.

6.1 Web Accessibility

Despite improvements in web accessibility over the past years, the web still does not fully comply with the accessibility standards set by the Web Content Accessibility Guidelines (WCAG) [36]. Numerous tools have been developed to tackle this issue and to support developers in the creation of accessible web pages. For instance, Axe-Core [33] is widely used in development pipelines and browser extensions because of its rule-based approach. Lighthouse [18] is another popular tool that is integrated into the Chromium-based browser and provides a set of accessibility checks as part of its performance audits. Other tools, such as WAVE [37], QualWeb [25], or Pa11y [8], offer similar functionalities and integrations for developers. Their shared objective is to automatically detect common accessibility violations directly within the development process by providing visual reports or highlighting problematic elements in the code.
The potential of LLMs to improve or even automate accessibility evaluation has been investigated in recent developments in AI-assisted accessibility [6, 12, 28]. For instance, Lopez-Gil and Pereira [26] explored the use of Chat-GPT and GitHub Copilot to detect accessibility issues, but their approach is limited to only 3 WCAG guidelines. He, Huq, and Malek [20] proposed a framework, called *GenA11y*, which uses LLMs to extract page elements relevant to 37 WCAG guidelines and create prompts to detect possible violations. This prior research shows the potential of LLMs to bridge gaps in existing accessibility tools, particularly in the area of automated code generation. Even though they have marked an important step towards a more accessible web, they still face limitations. For instance, they often only offer a limited set of WCAG guidelines and do not cover a wide range of accessibility issues. Additionally, they require a certain level of engineering effort and would be difficult to integrate into an empirical study at scale.

6.2 MLLMs for UI code generation

The use of MLLMs for UI code generation from visual inputs, known as *image-to-code* or *UI-to-code* generation, has been a topic of research in recent years. Early attempts, like *REMAUI* [29], focused on code generation from screenshots through traditional image processing techniques and heuristic rules. Later, more advanced approaches, such as *pix2code* [4], integrated computer vision (CV) and deep learning techniques. This approach used a convolutional neural network (CNN) to extract features from a screenshot and a recurrent neural network (RNN) to generate a frontend-specific language.
Within recent years, LLMs have improved their performance and capabilities in coding-related tasks. This has led to improved code generation, completion, and translation across different

programming languages and software engineering domains. A prominent example is Cursor [21], which is an LLM-based code editor that supports developers in daily coding tasks. The combination of LLMs and programming environments with tools like Cursor has demonstrated the practical impact of LLMs in real-world development scenarios.

The advancements in vision capabilities of MLLMs have enabled new possibilities for image-to-code generation. A prominent example is *DCGen* [35], a pipeline that segments screenshots into smaller visual segments and then generates code snippets for each of them. Those snippets are then reassembled to create the final code. Another example is *DeclarUI* [41], which improves the generation of UI code by using self-refinement prompt engineering techniques. This approach lets the models first generate a code snippet, then critique it, and finally refine it in an iterative process.

Despite this progress, the evaluation of accessibility of MLLM-generated code remains an underexplored area. One of a few studies by Aljedaani, Habib, Aljohani, et al. [1] evaluated ChatGPT's ability to generate accessible web content. They conducted a study which involved web developers who used natural language descriptions to instruct ChatGPT to generate website code. Their study demonstrated that 84% of generated sites contained accessibility violations.

However, these studies have limitations, e.g., they relied on natural language descriptions, which limits the applicability to real-world scenarios. Nowadays, visual design artifacts, such as screenshots or design mockups, are often the primary input. Therefore, this thesis focuses on UI screenshots as input for MLLMs, which makes it possible to evaluate the models' capabilities to work out semantic structures and accessibility features. This study is the first empirical study to systematically evaluate the accessibility of MLLM-generated code from visual inputs, while also taking the diversity of prompting techniques and model types into account. The findings of this thesis fill a gap in the existing literature and might provide actionable insights for future directions.

7 Conclusion

This thesis has first systematically evaluated the accessibility of HTML/CSS code generated by state-of-the-art multimodal large language models (MLLMs) from UI screenshots. While the models have shown impressive capabilities in the generation of syntactically valid code that closely mirrors the original UI image, they all reveal persistent accessibility violations. Through an empirical study and analysis across different models, prompting techniques, and datasets, this thesis identifies recurring WCAG 2.1 violations, demonstrates the limited impact of naive prompting, and shows how more advanced prompting techniques can reduce but not fully resolve the accessibility gap.

These findings highlight a critical insight: improving visual and functional code fidelity does not necessarily lead to accessible code. Instead, accessibility must be elevated to a first-class objective in the design and evaluation of AI-driven UI development. As MLLMs continue to evolve, the challenge and opportunity lie in integrating accessibility as a core principle into the models' architecture and training corpus. By rethinking accessibility as a fundamental necessity, and not just as a post-processing step, we can take a significant step towards a more inclusive and user-centered web.

List of Figures

List of Tables

Bibliography

[1] W. Aljedaani, A. Habib, A. Aljohani, M. M. Eler, and Y. Feng. "Does ChatGPT Generate Accessible Code? Investigating Accessibility Challenges in LLM-Generated Source Code." In: *Proceedings of the 21st International Web for All Conference (W4A '24)*. W4A '24. New York, NY, USA, 2024, pp. 165–176. DOI: 10.1145/3677846.3677854.

[2] A. Alshayban, I. Ahmed, and S. Malek. "Accessibility Issues in Android Apps: State of Affairs, Sentiments, and Ways Forward." In: *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 1323–1334. DOI: 10.1145/3377811.3380392.

[3] Be My Eyes. *Be My Eyes – See the world together*. Accessed: 2025-08-10. 2025.

[4] T. Beltramelli. "pix2code: Generating Code from a Graphical User Interface Screenshot." In: *arXiv preprint arXiv:1705.07962* (2017). DOI: 10.48550/arXiv.1705.07962. arXiv: 1705.07962.

[5] *Bootstrap v4 Documentation: Theming Bootstrap*. https://getbootstrap.com/docs/4.0/getting-started/theming/. 2018.

[6] E. Cali, T. Fulcini, R. Coppola, L. Laudadio, and M. Torchiano. "A Prototype VS Code Extension to Improve Web Accessible Development." In: *2025 IEEE/ACM Second IDE Workshop (IDE)*. IEEE. 2025, pp. 52–57.

[7] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu. "From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation." In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 665–676.

[8] P. Contributors. *Pa11y is your automated accessibility testing pal*. Version 9.0.0. 2025.

[9] Deque. *Deque Study Shows Its Automated Testing Identifies 57 Percent of Digital Accessibility Issues, Surpassing Accepted Industry Benchmarks*. https://www.deque.com/blog/automated-testing-study-identifies-57-percent-of-digital-accessibility-issues. Accessed: 2025-07-10. 2021.

[10] *Digital Silk. How Many Websites Are There In 2024?* https://www.digitalsilk.com/digital-trends/how-manywebsites-are-there/. 2025.

[11] *Disability*. https://www.who.int/news-room/fact-sheets/detail/disability-and-health. 2025.

[12] C. Duarte, M. Costa, L. Seixas Pereira, and J. Guerreiro. "Expanding Automated Accessibility Evaluations: Leveraging Large Language Models for Heading-Related Barriers." In: *Companion Proceedings of the 30th International Conference on Intelligent User Interfaces*. 2025, pp. 39–42.

[13] European Parliament and the Council of the European Union. *Directive (EU) 2019/882 on the Accessibility Requirements for Products and Services (European Accessibility Act)*. 2019.

[14] S. Feng. *What's Missing in Accessibility Testing Tools? Looking for Feedback and Ideas [closed]*. UX StackExchange Discussions post. June 2025. URL: https://ux.stackexchange.com/questions/153763/what-s-missing-in-accessibility-testing-tools-looking-for-feedback-and-ideas.

[15] S. Feng, M. Jiang, T. Zhou, Y. Zhen, and C. Chen. "Auto-icon+: An automated end-to-end code generation tool for icon designs in ui development." In: *ACM Transactions on Interactive Intelligent Systems* 12.4 (2022), pp. 1–26.

[16] N. Ferhan. *E-Commerce Site*. 2025.

[17] GitHub. *GitHub Copilot – Your AI Pair Programmer*. 2025.

[18] Google. *Lighthouse – Automated auditing, performance metrics, and best practices for the web*. Version 12.8.0. 2025.

[19] Y. Gui, Z. Li, Y. Wan, Y. Shi, H. Zhang, Y. Su, B. Chen, D. Chen, S. Wu, X. Zhou, W. Jiang, H. Jin, and X. Zhang. "WebCode2M: A Real-World Dataset for Code Generation from Webpage Designs." In: *arXiv preprint arXiv:2404.06369* (2024). DOI: 10.48550/arXiv.2404.06369. arXiv: 2404.06369v2.

[20] Z. He, S. F. Huq, and S. Malek. "Enhancing Web Accessibility: Automated Detection of Issues with Generative AI." In: *Proceedings of the ACM on Software Engineering* 2.FSE (2025), pp. 2264–2287.

[21] A. Inc. *Cursor – The AI Code Editor*. 2025.

[22] W. Inc. *Windsurf - The most powerful AI Code Editor*. 2025.

[23] W. W. A. Initiative. *ARIA Authoring Practices Guide (APG)*. https://www.w3.org/WAI/ARIA/apg/patterns/. 2025.

[24] A. O. Labs. *Alpha One Labs Education Platform*. Version v1.0. 2025.

[25] U. o. L. LASIGE. *QualWeb*. 2025.

[26] J.-M. Lopez-Gil and J. Pereira. "Turning manual web accessibility success criteria into automatic: an LLM-based approach." In: *Universal Access in the Information Society* 24.1 (2025), pp. 837–852.

[27] M. Lutz. *What's missing in accessibility testing tools?* Ministry of Testing Discussions post. July 2025. URL: https://club.ministryoftesting.com/t/what-s-missing-in-accessibility-testing-tools-looking-for-feedback-and-ideas/85633.

[28] P. Mowar, Y.-H. Peng, J. Wu, A. Steinfeld, and J. 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*. 2025, pp. 1–15.

[29] T. A. Nguyen and C. Csallner. "Reverse engineering mobile application user interfaces with remaui (t)." In: *2015 30th IEEE/ACM international conference on automated software engineering (ASE)*. IEEE. 2015, pp. 248–259.

[30] G. D. Service. *GOV.UK Design System: Components*. https://design-system.service.gov.uk/components/. 2025.

[31] C. Si, Y. Zhang, R. Li, Z. Yang, R. Liu, and D. Yang. "Design2Code: Benchmarking Multimodal Code Generation for Automated Front-End Engineering." In: *arXiv preprint arXiv:2403.03163v3* (2024). DOI: 10.48550/arXiv.2403.03163. arXiv: 2403.03163v3.

[32] H. Suh, M. Tafreshipour, S. Malek, and I. Ahmed. "Human or LLM? A Comparative Study on Accessible Code Generation Capability." In: *arXiv preprint arXiv:2503.15885* (2025). DOI: 10.48550/arXiv.2503.15885. arXiv: 2503.15885.

[33] D. Systems. *axe-core: Accessibility engine for automated Web UI testing*. Version 4.10.3. 2025.

[34] United States Congress. *Americans with Disabilities Act*. 1990.

[35] Y. Wan, C. Wang, Y. Dong, W. Wang, S. Li, Y. Huo, and M. R. Lyu. "Automatically Generating UI Code from Screenshot: A Divide-and-Conquer-Based Approach." In: *arXiv preprint arXiv:2406.16386* (2024). DOI: 10.48550/arXiv.2406.16386. arXiv: 2406.16386v3.

[36] *Web Content Accessibility Guidelines (WCAG) 2.1*. World Wide Web Consortium. May 6, 2025. URL: https://www.w3.org/TR/WCAG21/ (visited on 06/16/2025).

[37] WebAIM. *WAVE Web Accessibility Evaluation Tools*. 2025.

[38] F. Wu, C. Gao, S. Li, X.-C. Wen, and Q. Liao. "MLLM-Based UI2Code Automation Guided by UI Layout Information." In: *Proceedings of the ACM on Software Engineering* (2025). DOI: 10.1145/3728925.

[39] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang. "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation." In: *arXiv preprint arXiv:2308.08155* (2023). DOI: 10.48550/arXiv.2308.08155.

[40] J. Xiao, Y. Wan, Y. Huo, Z. Wang, X. Xu, W. Wang, Z. Xu, Y. Wang, and M. R. Lyu. "Interaction2Code: Benchmarking MLLM-based Interactive Webpage Code Generation from Interactive Prototyping." In: *arXiv preprint arXiv:2411.03292* (2024). DOI: 10.48550/arXiv.2411.03292. arXiv: 2411.03292.

[41] T. Zhou, Y. Zhao, X. Hou, X. Sun, K. Chen, and H. Wang. "Bridging Design and Development with Automated Declarative UI Code Generation." In: *arXiv preprint arXiv:2409.11667* (2024). DOI: 10.48550/arXiv.2409.11667. arXiv: 2409.11667v1.