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

Abstract

Since Multimodal Large Language Models (MLLMs) increasingly support UI code generation from visual inputs, such as UI screenshots, their role 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 four 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, and (4) 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 [8].
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) [23] 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 the over one billion people globally who live with some form of disability [9].
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 [21] and the European Accessibility Act (EAA) in the EU [10]. Neglecting to follow these regulations, could result in warnings, reputational damage and loss of sales in the future.
At the same time, Large-Language Models (LLMs) have demonstrated signifant improvements in code-related tasks, including code generation, completion and summarization. Current Tools, such as GitHub Copilot [12], Cursor [15], Windsurf [16], are capable to support 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 [5, 11]. 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 [4, 17, 25], leveraging MLLMs to better capture layout structures, semantics and component hierarchies. For instance, DCGen [22] uses a divide-and-conquer pipeline to segment UI screenshots and then generate code for each segment respectively. DeclarUI [28] integrates MLLMs with advanced prompting strategies, particullary a self-refinement loop in which a multimodal model reviews and revises its draft to improve code quality.
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 [19] compared LLM-generated code with human counterparts in terms of 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.

1.1.1 Research Questions

The investigation is based on the following research questions:
RQ1: Do MLLMs generate accessible code from UI screenshots? This question investigates wether 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 evaluated using automated accessibility auditing tools and manual analysis to identify potential violations of the WCAG 2.1 guidelines. This question seeks to uncover wether 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 size 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). Using the same benchmark dataset and naive prompting approach, the numbers and types of accessibility violations for each model are compared. This comparison allows to identify the differences in model behavior and analyze potential sources of bias or limitations in accessibility compliance. A qualitative analysis of the generated code further explores how factors such as training data biases, model instructions and internal reasoning abilities contribute to the variance in performance across different models.

RQ3: Does advanced prompt engineering and (post-)processing steps lead to more accessible MLLM-generated UI code? This question explores wether advanced prompting strategies can guide MLLMs to generate 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, agentic prompting where multiple agents split the tasks of detecting, classifying and solving violations. Lastly, two strategies use external tools to enhance their output: ReAct prompting where the model iteratively critiques and improves its own output based on violations found by accessibility tools, color-aware prompting uses ReAct to critique its own violations, but further instructs the model with color contrast information and potential fixes. Those strategies are evaluated based on the same benchmark dataset across the different MLLMs and resulting violations are assessed. This investigation reveals the effectiveness of prompting as a controllable factor in improving accessibility violations, also highlighting potential side effects, such as cascading errors introduced during refinement steps. Those finding provide insights for developer and reseachers aiming to guide MLLMs towards more inclusive code generation.

RQ4: 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 three distinct datasets: the public benchmark dataset used in previous RQs, a synthetic dataset created through structural mutations and 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 violations across these datasets, this questions tries to identify wether 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 empirical study demonstrates both the potential and limitations of current MLLMs in generating accessible UI code from visual inputs. Motivated by these findings, this thesis reflects on broader implications for the design and deployment of generative models in web development. Especially, it intends to rethink accessibility as a primary design objective, rather than a post-hoc concern. These perspectives offer possible directions for enhancing accessibility in the era of multimodal code generation.

1.1.2 Our Contributions

In summary, this thesis makes the following contributions:

Accessibility evaluation pipeline The first large scale accessibility study and evaluation pipeline of LLM-based Image-to-Code generation is proposed. This pipeline combines visual and structural fidelity with an automatic WCAG 2.1 conformity check.

Realistic dataset This study uses a realistic dataset that contains of 53 real-world webpage examples which have been gathered from existing datasets and slightly mutated to minimize noise within the data. It covers a wide spectrum of layouts, content areas and accessibility features. This dataset combines the screenshots and HTML/CSS code of each webpage.

Model and prompting comparison This study compares multiple MLLMs and prompting strategies across diverse benchmarks and conducts a qualitative analysis to understand their impact on the accessibility of generated code. We release our experimental dataset, the code and the results on Github[1].

In-depth quantitative and qualitative discussion This study presents an in-depth discussion of our findings and proposes future directions for improving MLLM-driven UI-workflows.

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

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 ensures to identify a wide range of accessibility issues, reconcile inconsistencies across tools and guarantee reliable results. All evaluations are conducted in accordance with WCAG 2.1, which is 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 are available to detect violations in web content. According to former studies, automated testing can detect up to ∼60% of accessibility issues [7]. This makes them valuable for developers, however they can not replace manual testing or expert review completely. However in practice, a combination of various tools can help to 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) [20]:** is a rule-based engin developed by Deque Systems and commonly integrated into browser extensions and CI/CD pipelines. It provides detailed issue classifications and links each violation to an own rule-set.
- **Google Lighthouse Accessibility (12.4.0) [13]:** embedded within Chromium-based browsers, performs accessibility audits alongside performance and SEO diagnostics.
- **Pa11y (8.0.0) [6]:** 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, we found out that Pa11y has especially strengths in areas where the other tools seem to fail. Therefore, we decide to use it as our third complementary tool.

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, that 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. In order to aggregate the outputs of the three tools, first all detected violations are compiled. 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, overlapping rule definitions are merged 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 unified under 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 replace manual testing completely. 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(S) (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 body 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 the 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-worldwebpages. 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. There is no publicly available dataset which fullfills all requirements.

2.3 Construction

Two promising examples in the field of Image-to-Code are *Design2Code* [18] and *Webcode2m* [14]. 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 instance from WebCode2M, resulting in a total of 53 UI–code pairs, is used for the study. This sampling strategy allows for representative, yet feasible evaluation given the resource constraints of this thesis.

Content Distribution

Figure 2.1 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.

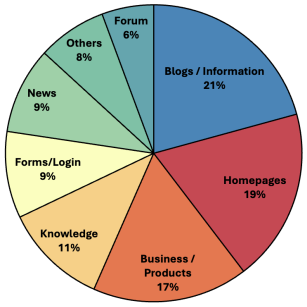


Figure 2.1: Distribution of Topics in Dataset

2.3.1 Dataset Alignment

Because Design2Code and Webcode2m use different strategies to purify their data, we harmonize the datasets prior to the analysis. The alignment contains (1) removing all external dependencies (e.g. images, audio, video, external links) and substituting neutral placeholders (e.g. `src="placeholder.jpg"` for images or `href="#"` for links); (2) disabling 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 otherwise possibly bias the accessibility metrics.

2.4 Benchmarks

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

2.4.1 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 [18], the evaluation is based on a combination of high-level visual conformity and low-level element matching, capturing 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: $CodeSim = \sum_{i=1}^{5} w_i \cdot S_i,$, 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 to get a balanced view of the visual and structural similarity of the input and output.

2.4.2 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 not only decrease the amount of accessibility violations, but also its 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:

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

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:

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

3 Related Work

3.1 Background

Large Language Models (LLMs) and their performances in various domains are improving rapidly. Especially, in the domain of code generation those models show promising results. It is therefore not surprising to see attempts to automate the creation of webpages and frontend code.

3.2 Image-to-Code

The focus of the first attempts in this area was to capture the image as precise as possible in order to translate it into Frontend Code. For instance, *pix2code* [3] used a combination of CNN encoder with a LSTM decoder to translate screenshots into a frontend specific language. While it showed promising results for the possibility of end-to-end learning, it could not create standard HTML/CSS.
Within the recent years, LLMs have improved a lot and new vision capabilities have been added to the models. Instead of further retraining models, researchers have explored the capabilities of different prompting structures and pre-processing steps. A prominent example is *DCGen* [22] where researchers have segmented screenshots into smaller, visual segments for the LLMs to generate code for each segment and reassemble them afterwards. This approach reduces the misplacement of components and shows improvements in the visual similarity.
Other related papers explored ways to improve prompting techniques (paper). They showed that advanced prompting techniques, such as few-shot, chain-of-thought and self-reflection can improve the performance without changing the models parameters.

3.3 Web Accessibility

Even though the web has become more accessible over the past years, almost every website does not fully comply with the Web Content Accessibility Guidelines (WCAG) [23]. According to the 2025 annual *WebAIM* accessibility report, an average of 51 errors per webpage has been (noch aufnehmen, dass 96% verstöße) found across one million webpages tested [24].
In order to tackle this issue, recent research has inspected this topic. First, Aljedaani, Habib, Aljohani, et al. [1] asked developers to let ChatGPT generate frontend code and observe the corresponding accessibility violations of the outputs. While they found out that 84% of the webpages contained accessibility violations, they also demonstrated the LLMs' capabilities to repair roughly 70% of its own mistakes. However, more complex issues remain. Similar results have been shown by Suh, Tafreshipour, Malek, and Ahmed [19]. Introducing a feedback-driven approach helped to further improve the WCAG compliance.
While recent research shows promising results, there does not exist a comparable work in the area of Image-to-Code.

3.4 AI-enhanced GUI testing

if necessary...

4 Empirical Study

To better understand the capabilities and limitations of MLLMs in generating accessible UI code, a comprehensive study guided by four research questions was conducted. In the following, the study setup and the quantitative and qualitative results are presented.

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

4.1.1 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 make 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 solely based on the provided screenshot, without mentioning accessibility.

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

This setups 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 presented represent the average outcomes to account for stochastic fluctuations in the model's responses.

4.1.2 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 4.1. 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 4.1 presents the amount of accessibility violations in UI code generated by GPT and Gemini. Despite achieving high layout and structural fidelity, both models produce a significant number of accessibility issues. On average, GPT generates 13.75 violations per UI, while Gemini produces 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 amount 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 highlights that without clear and explicit instructions, MLLMs might not be able to follow accessibility principles in code generation. In order to understand the underlying reasons for these accessibility issues, a manual analysis of the violations observed in the

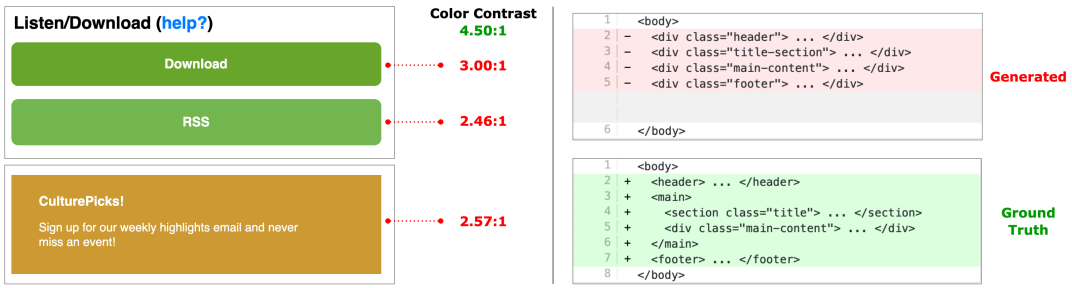


Figure 4.1: Example of common accessibility violations.

generated output has been conducted. This process allows to identify recurring failure patterns and model limitations.

Most common violations

The detailed analysis reveals that two primary categories of accessibility issues are prevalent in the generated code across both models. The first category is insufficient color contrast. For instance, color contrast violations are very common, producing 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, 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 4.1 (a), the generated code uses a white text (#ffffff) on a green background (#67a52a and #72b650), or on a yellow background (#cc9933), which 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 absence or incorrect use of semantic landmarks. Semantic landmarks, such as `main`, `nav`, `header`, and `footer` are essential for transforming the visual structure of a webpage into a format used by assistive technologies. 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 4.1 (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, lacking navigational and structural cues which are necessary for keyboard users or screen reader navigation. This can significantly hinder the accessibility, especially for users who rely on semantic segmentation to jump between regions of a page. Essentially, the problem arises, first, due to the absence of visible indicators, second, because of the model's limited exposure to the underlying meaning of UI elements—the type of information human designers tend to acquire through accessibility trainings 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 4.1: Performance for accessibility violations within the MLLMs-generated UI code.

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

4.2.1 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 employed to capture significant variation in model behavior and performance while ensuring practicability 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.

4.2.2 Results

Table 4.1 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 is 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 4.2: Comparison of semantically rich (left) and structurally shallow (right) HTML snippets

to the 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, 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 **??**. 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 which 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, they tend 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 4.2 (a), GPT tends to produce descriptive alt-text and labels by default.
While the reasons remain speculative, a plausible explanation could lie in the accessibility-oriented use cases and alignment practices by GPT. For example, GPT has been used in real-world assistive technologies like *Be My Eyes*. 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.

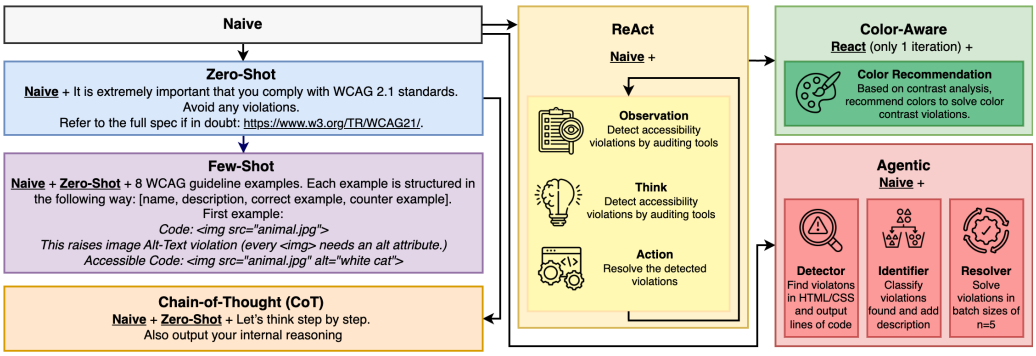


Figure 4.3: Overview of the advanced prompting strategies.

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 4.2 (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.

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

4.3.1 Experiment Setup

While the prior RQs explored the inherent capabilities of MLLMs with naive prompting, this RQ investigated the promptings 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 [19, 27].

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

- 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 Fig. 4.3. 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) [**web:w3c_examples**] and Accessible Components by Gov.uk Design System [**web:govuk**], 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. 4.3, 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 [26] 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 enrich 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 critique its accessibility violations detected by the automated tools. As shown in Fig. 4.3, 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 which 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 basing the model on 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.

Note that all prompting techniques are tested on GPT and Gemini by using the same benchmark dataset.

4.3.2 Results

Table 4.2 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 amount 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 the naive counterpart. A similar trend can be observed for the IR and IWIR. While the IR is reduced to 0.68% 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 4.2: 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 4.4 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.

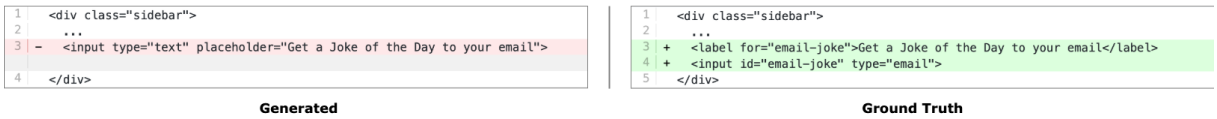


Figure 4.4: Failure example for zero-shot prompting.

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 and the IR decreasing from 12.22% to 7.96%. By observing 8 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 important accessibility guidelines, having a significant impact on users and therefore also a high severity. This is reflected in the IR, 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, in fact, 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 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 and links, such as the primary blue from the Bootstrap framework, that is used as the main color for links. This leads to a lack of distinction between links and the body text.
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 deeper semantic understanding in how those elements contribute to accessibility.

Impact of Chain-of-Thought Prompting

Chain-of-Thought 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 constaints 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 due to the fact that 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 and from 11.42% to 8.36% for Gemini, 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 in 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 which 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 3 iterations, only 0.44% of nodes in GPT's code contain accessibility violations, while Gemini has an IR of 1.36%. Interestingly, not only the amount of violations, but also their severity is 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 3 refinement iterations, neither GPT nor Gemini produces any violations with severity *critical* and only few with *serious*.
The results suggest that even if models 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 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, also the ReAct prompting 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 suggests that the model might not be able to synthesize its own new colors, but rather tries to substitute problematic colors with known colors from its training data. Furthermore, some attempts to fix particular violations can occasionally lead to new problems elsewhere in the code. As Fig **??** suggests, the model ... . These cascading error may arise from a limited understanding of the codes 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 attrubuted to this finding. First, GPT is already able to synthesize new colors. Providing new externally computed colors, occassionally 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 the 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 determines how beneficial this strategy is.

5 Evaluation

5.1 Accessibility Results

In order to provide a comprehensive analysis of the amount and type of accessibility violations, we divide the analysis into 3 categories. We report both a quantitive with aggregate metrics and a qualitive analysis with fine-grained explanations.

5.1.1 Quantitive Analysis

Prompting Techniques

Figure xy illustrates the comparison of the average amount of violations, the Inaccessibility Rate (IR) and the Impact-Weighted Inaccessibility Rate (IWIR) between the human baseline and the models with the corresponding prompting techniques.

Key observations: Even the weakest LLM outperforms the human baseline regarding the average amount of violations per webpage and the IR. Only the IWIR remains constant, only showing small differences across the models. The human-written HTML/CSS of our dataset counts 1339 accessibility violations, leading to $\sim$25.26 violations per file across the whole dataset. On the other hand, even gemini with the naive prompting technique, the worst performing set of parameter, had a maximum of 917 accessibility violations, leading to $\sim$17.3 violations per file across the whole dataset.
GPT-4o achieved the lowest average amount of violations per webpage, IR and IWIR.
Advanced prompting techniques show only little effect, compared to the naive baseline, demonstrating the LLMs' inherent understanding of accessibility. Even if the naive prompting approach does not instruct the LLMs to generate code with compliance to the WCAG standards, it still only shows slightly increased amounts of violations than more advanced prompting techniques.

Error Distribution: Figure yxc illustrates the distribution of violations per WCAG success criterion. The distribution shows a similar left-skewed distribution across all models, indicating that the models have a similar understanding of the WCAG rules.
At least 65% of all violations are caused by color contrast and landmark and region issues. This finding is not only consistent across the different models and prompting techniques, but also in human-written code.
The following types of violations are mainly caused by missing labels, wrong link colors, issues with header tags and the size of frontend components. This demonstates that only a small subset of WCAG violations have relevant and non-negligible amount of violations. Especially GPT-4o shows illustrates this clearly, as only 6 types of violations cause 94%(number check) of all violations.
The results also show differences between the models. While gemini seems to have more color contrast violations, landmark and region rules cause more problems for the GPT-4o model.

Table 5.1: Accessibility benchmarks: Inaccessibility Rate (IR), Impact-Weighted Inaccessibility Rate (IWIR), Average Number of Violations per webpage (ANV) and Total Violations (TV).

Technique	Gemini Flash 2.0				ChatGPT-4o				Qwen2.5vl-7B			
	IR	IWIR	ANV	TV	IR	IWIR	ANV	TV	IR	IWIR	ANV	TV
Naive	0.1142	0.4770	15.45	819	0.1222	0.4710	13.75	729	0.1070	0.4025	6.21	329
Zero-Shot	0.1033	0.4774	14.25	755	0.1002	0.4735	12.33	653	0.1094	0.3906	5.53	293
Few-Shot	0.1093	0.4791	15.74	834	0.0796	0.4464	9.49	503	0.0823	0.4092	5.92	314
CoT	0.1107	0.4831	14.28	757	0.0861	0.4453	10.04	532	**0.0689**	0.4599	6.49	344
IterativeRef1	0.0308	0.3872	5.85	310	0.0173	0.3188	2.43	129	0.0956	0.3513	5.79	307
IterativeRef2	0.0175	0.2596	3.45	183	0.0076	0.2128	1.09	58	0.0830	0.2875	5.17	274
IterativeRef3	**0.0136**	**0.2148**	2.77	147	**0.0044**	**0.1356**	**0.68**	**36**	0.0768	**0.2775**	5.04	267
Composite	0.0174	0.3493	**2.58**	**137**	0.0235	0.3368	3.34	177	0.0898	0.4044	**4.96**	**263**
Agent	0.0836	0.4700	13.08	693	0.0886	0.4180	11.28	598	0.0997	0.4042	6.08	322
Human Baseline	**0.1131**	**0.565**	**25.26**	**1339**	**0.1131**	**0.565**	**25.26**	**1339**	**0.1131**	**0.565**	**25.26**	**1339**

5.1.2 Qualitive Analysis

Consistency

We model the consistency of violations found within different experiment runs and dataset entries. Therefore, we use the *cosine-similarity* for each k-dimenstional error vector, where each dimension represents the amount of a specific WCAG violation. The cosine-similarity is then calculated between the vectors of the different experiment runs and dataset entries. As the heatmap in figure yys illustrates, light colors are dominating the tiles, indicating a high cosine similarity (mü = 0.9). This indicates that the LLMs do not only produce similar accessibility violations, but also that the amount of each type violation is consistent across each input webpage. Darker tiles coincide with lower cosine similarites. The majority of those darker tiles are mainly caused by webpages with only a few violations. For those webpages, halluzinations or randomly created mutations by the LLMs, such as new colors or missing landmarks, have a larger impact on the cosine similarity, as the distribution of violations is not consistent.

Concentration of Violations

While we do not know the exact training data of each model, we can infer by the observations that the models have been trained on similar underlying data. This also alligns with the error distribution of human developers, as WebAIM's 2025 "Million" report shows. In this report, 79% (number check) of webpages fail the color contrast guidelines, while 43% (number check) of webpages do not use landmarks correctly. Combined with the other WCAG rules that can be found in non-negligible amounts, they are considered more complex and require a deeper understanding of the underlying HTML/CSS structure.
This bias is faithfully reflected in the LLMs' results, which gives us confidence to hypothesize that scraped code from forums like StackOverflow inforced this shortcut in the models' weights. Answers on forums often only start with the first element, not showing the full page structure. This could explain the incorrect usage of landmarks and regions.

1. **Amount of Issues** For each test run and file we are counting the number of violations

per class

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

2. **Calculation of Cosine Similarity** Given two experiment runs with $\mathbf{x}, \mathbf{y} \in \mathbb{R}^k$, we define

$$\text{cos\_sim}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x}^\mathsf{T}\mathbf{y}}{\|\mathbf{x}\| \, \|\mathbf{y}\|} \quad \in [0, 1].$$

This cosine similarity is then plotted into a heatmap comparing the different experiment runs and files. The results can be seen in figure ab below. It is noticeable that most of the tiles show a bright color, referring to a high cosine similarity. The tiles with a darker color are mainly caused by 2 reasons. The first reason are files with only little violations that cause smaller cosine similarities due to the non-consistent distribution of violations. The second reason are randomly-generated files which have been mutated in such a way that they chose colors that comply with the Color Contrast Rules. Since overall the color constrast issues are one of the most common issues, this leads to a lower cosine similarity.
Those findings are consistent different models and prompting techniques. This demonstrates that the accessibility issues are consistent across multiple runs and not caused by halluzination of the LLMs but are based on their training data and the underlying parameters.
In a last step, the question arises why we see different results and violation distributions across the different models. Even though current LLMs are a *black-box* regarding their training data and its impact, we can infer some possible bias based on recent accessibility studies. As the *WebAIM 2025 Million* report [24] shows, 79% of all webpages contain low-contrast text. Similar results can be seen for WCAG landmark and region violations. 80,5% of webpages contained at least one region, but only for 42,6% a <main> element was present in the code. Other possible web-crawl training data sources like *StackOverflow* and other forums could further bias the LLMs since they often start with the first and omit the full page structure. This training data bias could explain the observed differences in the amount and type of accessibility violations. Lastly, many of the observerd violations, such as color contrast can be classified as more sophisticated, requiring a LLM to focus on the relative luminance and color contrast ratio. On the other hand, correct landmarks and regions require invisible semantics, apart from the raw pixel input of images. This semantic gap of information also requires deeper reasoning by the models. In conclusion, the observed violations follow a human bias which come from the vast training data that does not adhere fully to accessibility best practices.

Dominance of Default Colors

A further fine-grained analysis of the violations (Table xy) reveals that the majority of color contrast violations are caused by a small subset of colors. After inspecting the most common colors, those colors can be classified as *default colors*. They are often used by browsers and popular frameworks and thus have been learned by the LLMs. For instance, we identified two public color palettes - the *Google Color Palette* and the *Bootstrap v3-v5* - that explain a majority of the violation.

- **Gemini.** 13 out of 127 colors that violate the color contrast rules matches exactly one of the two color palettes. They are accounting for 65% of all misses, 71% if black (#000000) and white (#ffffff) are included.

- **GPT-4o.** 10 out of 237 colors match, leading to 24% of all violations (39% if black and white are included).

Gemini appears to use a larger amount of boiler-plate colors, for instance using Bootstrap's link color #007bff and its grey scale colors such as #777777 - #999999.
On the other hand, apart from some default colors, GPT-4o uses a wider variety of colors. It appears that GPT-4o is more likely to mix its own colors into the generated code, rather than relying on boiler-plate colors. This hypothesis can be further supported by the fact that GPT-4o is able to solve a higher percentage of color contrast violations than Gemini.

Table 5.2: Most common colors used by the LLMs.

Model	Distinct Colors with Violations	Top-7 colors	Percentage of all color contrast violations
Gemini Flash 2.0	127	#ffffff, #777777, #007bff, #0000ee, #888888, #999999, #29abe2	75%
GPT-4o	237	#ffffff, #777777, #888888, #999999, #ff0000, #007bff, #00ffff	48%
Qwen 2.5-vl	0		tbd

5.2 Image-to-Code Similarity

Prompting Techniques

Advanced Strategies

Since Image-to-Code main task is to copy the input image as precise as possible, we have analysed the performance across the different parameter sets to see how exact their results remain. The results in table as in the appendix indicate that the final scores decrease slightly when further accessibility instructions are mentioned to the LLMs. While the text and position similarity remains constant, the size and especially the text color similarity scores decrease. This is caused due to accessibility compliance that can cause the LLMs to choose different colors and even component sizes to align with the WCAG issues. However, the changes in terms of the final score are very small and almost negligible.
Overall, similar to former research gpt-4o demonstrates the best performance in this field by outperforming gemini flash-2.0 by a few percent.

6 Conclusion

6.1 Section

ncoh bessere Überschrift finden

6.1.1 Future Directions

Möglicherweise manual catalog mit aufnehmen RAG (externes Wissen) Fine-Tuning

7 Appendix

7.1 Results Data Leakage

Table 7.1: OpenAI GPT-4o: Data Leakage (DL) based on 3 iterations

		Final Score	Size	Text	Position	Text Color	CLIP
DL Test Dataset	Naive	**0.8917**	0.8812	0.9701	0.8562	0.8451	0.906
	Zero-Shot	**0.8889**	0.866	0.9737	0.8543	0.8407	0.9098
	Few-Shot	**0.8929**	0.8929	0.9756	0.8486	0.8394	0.9078
	Reasoning	**0.8924**	0.8819	0.9755	0.8498	0.8449	0.91
	Iterative	**0.8908**	0.8819	0.9748	0.8475	0.8391	0.9109
	Iterative Refine 1	**0.8878**	0.8729	0.974	0.8469	0.8372	0.9081
	Iterative Refine 2	**0.8887**	0.8642	0.9771	0.8516	0.8439	0.9069
	Iterative Refine 3	**0.8871**	0.8497	0.979	0.8511	0.8483	0.9076
Experiment Dataset	Naive	**0.8896**	0.868	0.9661	0.8578	0.8456	0.9107
	Zero-Shot	**0.8779**	0.8124	0.9663	0.8558	0.8467	0.9083
	Few-Shot	**0.8729**	0.8131	0.9645	0.8562	0.8242	0.9067
	Reasoning	**0.8791**	0.8348	0.9652	0.8549	0.8358	0.9048
	Iterative	**0.8854**	0.8447	0.9694	0.8577	0.8412	0.914
	Iterative Refine 1	**0.8786**	0.8306	0.9677	0.858	0.8233	0.9131
	Iterative Refine 2	**0.8767**	0.8148	0.968	0.854	0.8315	0.915
	Iterative Refine 3	**0.8731**	0.811	0.9685	0.855	0.8181	0.9127

Table 7.2: Gemini-2.0-flash: Data Leakage (DL) based on 3 iterations

		Final Score	Size	Text	Position	Text Color	CLIP
DL Test Dataset	Naive	**0.8801**	0.7992	0.9685	0.8591	0.8251	0.9079
	Zero-Shot	**0.8798**	0.8297	0.977	0.8645	0.8141	0.9134
	Few-Shot	**0.8729**	0.8131	0.9645	0.8562	0.8242	0.9067
	Reasoning	**0.8683**	0.799	0.9741	0.8541	0.8093	0.905
	Iterative	**0.8823**	0.8298	0.9742	0.8624	0.836	0.9091
	Iterative Refine 1	**0.8783**	0.8297	0.9753	0.8616	0.8136	0.9112
	Iterative Refine 2	**0.8874**	0.8617	0.9774	0.871	0.8182	0.9086
	Iterative Refine 3	**0.8899**	0.8682	0.9773	0.8719	0.8224	0.9099
Experiment Dataset	Naive	**0.8712**	0.7992	0.9686	0.8591	0.8215	0.9079
	Zero-Shot	**0.8685**	0.7875	0.9687	0.862	0.8166	0.9094
	Few-Shot	**0.8695**	0.7989	0.9658	0.8627	0.8154	0.9048
	Reasoning	**0.868**	0.7916	0.963	0.8594	0.7996	0.9067
	Iterative	**0.8707**	0.7891	0.9686	0.8657	0.8209	0.9093
	Iterative Refine 1	**0.8622**	0.7703	0.9654	0.8616	0.8073	0.9064
	Iterative Refine 2	**0.8676**	0.7803	0.9683	0.8724	0.8132	0.9039
	Iterative Refine 3	**0.8609**	0.7708	0.9672	0.8707	0.7939	0.9017

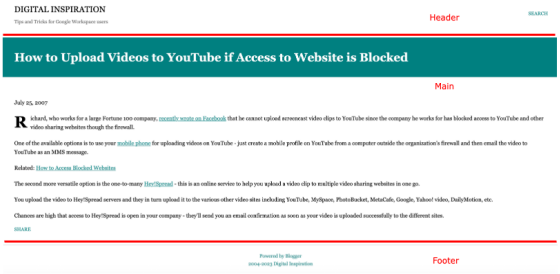
7.2 Results Code Similarity

Table 7.3: Results of Code Similarity for each model based on 3 runs.

		Final Score	Size	Text	Position	Text Color	CLIP
Gemini	Naive	**0.8712**	0.7992	0.9686	0.8591	0.8215	0.9079
	Zero-Shot	**0.8685**	0.7875	0.9687	0.862	0.8166	0.9094
	Few-Shot	**0.8695**	0.7989	0.9658	0.8627	0.8154	0.9048
	Chain-of-Thought	**0.868**	0.7916	0.963	0.8594	0.7996	0.9067
	Iterative Refine 1	**0.8622**	0.7703	0.9654	0.8616	0.8073	0.9064
	Iterative Refine 2	**0.8676**	0.7803	0.9683	0.8724	0.8132	0.9039
	Iterative Refine 3	**0.8609**	0.7708	0.9672	0.8707	0.7939	0.9017
	Composite	**0.8559**	0.774	0.9656	0.8596	0.7744	0.9059
	Agent	**0.8507**	0.7509	0.9631	0.8489	0.7901	0.9004
ChatGPT	Naive	**0.8896**	0.868	0.9661	0.8578	0.8456	0.9107
	Zero-Shot	**0.8779**	0.8124	0.9663	0.8558	0.8467	0.9083
	Few-Shot	**0.8729**	0.8131	0.9645	0.8562	0.8242	0.9067
	Chain-of-Thought	**0.8791**	0.8348	0.9652	0.8549	0.8358	0.9048
	Iterative Refine 1	**0.8786**	0.8306	0.9677	0.858	0.8233	0.9131
	Iterative Refine 2	**0.8767**	0.8148	0.968	0.854	0.8315	0.915
	Iterative Refine 3	**0.8731**	0.811	0.9685	0.855	0.8181	0.9127
	Composite	**0.8642**	0.8061	0.9666	0.853	0.7879	0.9072
	Agent	**0.8612**	0.781	0.9631	0.8456	0.8081	0.9081
Qwen	Naive	**0.7036**	0.6336	0.7755	0.6348	0.6313	0.8427
	Zero-Shot	**0.6875**	0.63	0.7561	0.6105	0.6032	0.8377
	Few-Shot	**0.747**	0.6813	0.8507	0.6809	0.6601	0.8619
	Chain-of-Thought	**0.8194**	0.7557	0.9545	0.7748	0.7454	0.8668
	Iterative Refine 1	**0.7083**	0.6081	0.7976	0.6422	0.6451	0.8484
	Iterative Refine 2	**0.7006**	0.5926	0.7896	0.6369	0.637	0.8467
	Iterative Refine 3	**0.7046**	0.584	0.7985	0.6451	0.6459	0.8494
	Composite	**0.7147**	0.6371	0.7985	0.644	0.6468	0.8472
	Agent	**0.7101**	0.6305	0.7873	0.6423	0.6462	0.8439

Table 7.4: Accessibility violations (absolute and mean per file) across prompting techniques and models.

		# Viol.	Mean/file
Gemini	Naive	432	2.10
	Zero-Shot	451	2.19
	Few-Shot	428	2.07
	Chain-of-Thought	439	2.12
	IterativeRef1	415	2.01
	IterativeRef2	408	1.98
	IterativeRef3	421	2.05
	Composite	399	1.93
	Agent	387	1.88
ChatGPT	Naive	378	1.84
	Zero-Shot	395	1.92
	Few-Shot	369	1.80
	Chain-of-Thought	382	1.86
	IterativeRef1	359	1.75
	IterativeRef2	351	1.72
	IterativeRef3	364	1.78
	Composite	346	1.68
	Agent	341	1.66
Qwen	Naive	612	3.04
	Zero-Shot	598	2.97
	Few-Shot	572	2.85
	Chain-of-Thought	529	2.64
	IterativeRef1	601	3.00
	IterativeRef2	609	3.05
	IterativeRef3	595	2.98
	Composite	583	2.92
	Agent	589	2.95



(a) Rendered webpage used as input

```html
<!-- generated by ChatGPT-4o -->
<!DOCTYPE html>
<html lang="en">
  <head> ... </head>

  <body>
    <div class="header"> ... </div>
    <div class="title-section"> ... </div>
    <div class="main-content"> ... </div>
    <div class="footer"> ... </div>
  </body>
</html>
```

(b) HTML output from Model ChatGPT-4o lacking semantic landmarks

Figure 7.1: ChatGPT-4o accurately reproduces the visual layout shown in (a), yet omits semantic landmark elements, such as `<header>` and `<main>` in the generated HTML (b)

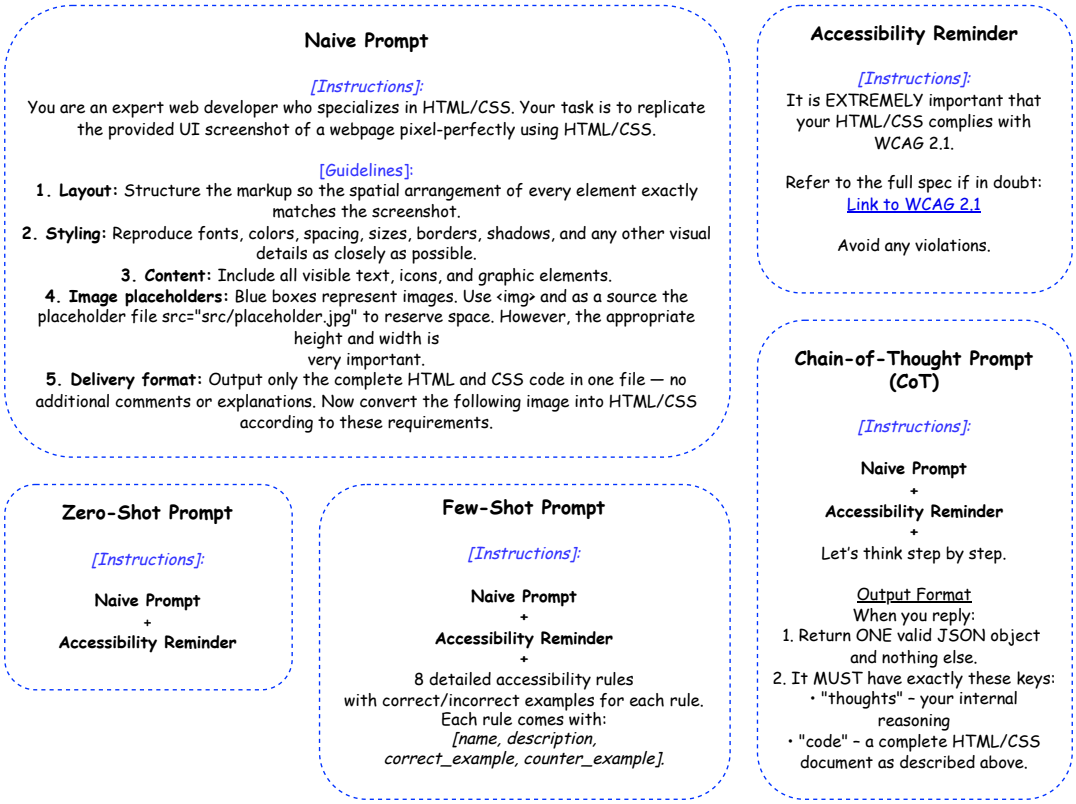


Figure 7.2: Overview of Prompts used.

Table 7.5: Mapped Accessibility-Violations

Technique	Success Criterion	Mapping-Name	Description
H30.2, H91.A	link-name	Links; Missing descriptive content of `<a>`	Link has no perceivable (visible/AT) name.
–	link-in-text-block	Links; Distinguishable Color	Only color distinguishes the link from body text.
H59	–	Links; Uncomplete	Incomplete or malformed `<a>` element.
H36, H37, H67	image-alt, input-image-alt	Alt-Text; Images	Image lacks alternative text.
H2	image-redundant-alt	Alt-Text; Images and Links; Redundant Alt-Text	Alt text simply repeats visible text.
H42, G141	page-has-heading-one, heading-order, empty-heading	Headings; Wrong Order, Empty and Missing Headings;	Headings missing, empty, or out of logical order.
–	landmark-one-main, landmark-unique, region, landmark-no-duplicate-contentinfo, landmark-no-duplicate-main, landmark-main-is-top-level	Landmark and Region; Missing and Unique Landmarks;	Landmarks missing, duplicated, or incorrectly nested.
H91, F68	label	Label; Multiple Elements; Content missing	Form controls missing or having multiple labels/content.
H91	input-button-name, button-name	Label; Button; Missing Label	(Button) control lacks an accessible name.
H93, H44, H65	form-field-multiple-labels	Label; Form Field; Multiple IDs, No ID, Wrong for attribute	Form field with multiple/incorrect labels or IDs.
H63	scope-attr-valid, td-headers-attr	Tables; Scope Attribute;	Incorrect scope/headers association in table.
H43, H63	empty-table-header	Tables; Table Headers;	Table headers missing, empty, or wrongly referenced.
–	td-has-header	Tables; Table Data must have Table Header;	Data cells missing an associated header.
G18, G145, G17	color-contrast, color-contrast-enhanced	Color Contrast; Text;	Insufficient text/background contrast.
H25.1	document-title	Title; Document Title;	Document title missing or empty.
H57, H58.1	html-has-lang, html-lang-valid, html-xml-lang-mismatch, valid-lang	Language; Document Lang; Missing and Invalid	HTML language missing or invalid.
H32	–	Elements; Form; No Submit Button	Form lacks a submit mechanism.
F77	–	ID; Duplicate IDs;	Duplicate `id` attributes.
–	target-size	Size; Target Size; Element too small	Interactive target too small.
–	aria-prohibited-attr	Aria Attributes; Prohibited Attributes;	Prohibited ARIA attribute used.
–	aria-valid-attr-value, aria-valid-attr	Aria Attributes; Valid Values;	Invalid ARIA attribute value.
–	aria-allowed-attr	Aria Attributes; Allowed Attributes;	ARIA attribute not permitted for role.
–	label-title-only, label-content-name-mismatch	Aria Attributes; Label Problems;	Label only in `title` or inconsistent.
H48	list, listitem	List; Incorrect Structure;	Improper list structure (ul/ol/li).
H49.Center	–	Elements; Center;	Obsolete `<center>` element.
H49.Font	–	Elements; Font;	Obsolete `<font>` element.
H49.AlignAttr	–	Elements; Align;	Presentational `align` attribute used.
G142	–	Zoom; Text Zoom up to 200%;	Text cannot scale to 200% without loss.

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] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

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

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] *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).

[24] WebAIM. *The WebAIM Million - An annual accessibility analysis of the top 1,000,000 home pages*. Accessed: 2025-06-16. 2025.

[25] 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.

[26] 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.

[27] 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.

[28] 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.