SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

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

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

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

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

Author: Marco Lutz
Supervisor: Sidong Feng
Advisor: 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

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. Irrespective of its content, such as shop, private blogs or news pages, each webpage is evaluated based on its usability. However, yet the creation of webpages or user interfaces (UIs) follows a similar and repetitive pattern. First UI designs are created with the help of web-design tools. Those UI designs present the foundation for software developers. In a second step, they are translated into functional UI code which tries to resemble the intended layout and structure as close as possible.

One essential, but yet frequently underestimated aspect of quality in this process is *accessibility*: According to the official *WCAG* guidelines, code must be perceivable, operable, understandable and robust. This allows users with visual, hearing or cognitive impairments, as well as users of assistive technologies, to follow the content of a webpage.

Complying with accessibility standards is not only an optional or moral aspect of web development, but it now has to follow regulatory boundaries. For instance the *European Accessibility Act* came into effect on June 28, 2025 and obliges any e-commerce or digital service in the EU to comply with those standards. Neglecting to do so could result in warnings, reputational damage and loss of sales in the future.

At the same time, thanks to Large-Language Models (LLMs) we experience signifant improvements in automatic code generation. Current LLMs are capable to perform *Image-to-Code* tasks where based on a UI design or screenshot, LLMs generate functional frontend code code. Recent research has demonstrated that especially for less complex webpages LLMs show decent performance [Si+24].

But can LLMs also generate accessible frontend code in a realistic image-to-code scenario? This question has hardly been investigated to date. Especially due to rising frontend code generation, it will be interesting to see how the WCAG compliance will influence the visual similarity.

1.1.1 Our Contributions

Evaluation Pipeline

In order to close this gap, we propose a large scale accessibility evaluation pipeline of LLM-based Image-to-Code generation. This pipeline combines visual and structural fidelity with an automatic WCAG conformity check. Therefore, we propose different benchmarks in order to measure the performance of the LLMs.

Realistic Dataset

We create 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 contains the screenshots of each webpage, but also the HTML/CSS created by human developers.

Model Comparison

We conduct an in-depth comparison of 3 state-of-the-art LLMs with vision capabilities (gpt-4o, gemini flash 2.0, qwen 7B vl) under the same experiment conditions. The LLMs are tested across different prompting strategies, as well as pre- and post-processing techniques.

Quantitative and Qualitative Evaluation

Each result will be analyzed based on a quantitative and qualitative evaluation. We analyze differences across the LLMs and outline their reasons.

Best-Practice Guidelines

Based on our experience, we present concrete best-practice guidelines, how to combine different techniques in order to achieve a maximum amount of accessibility compliance.

2 Related Work

2.1 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.

2.2 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* [Bel17] 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* [Wan+24] 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.

2.3 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) [24]. 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 [Web25]. In order to tackle this issue, recent research has inspected this topic. First, Aljedaani, Habib, Aljohani, et al. [Alj+24] 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 [Suh+25]. 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.

2.4 AI-enhanced GUI testing

if necessary...

3 Dataset

3.1 Scope and Design

The main goal is to gather a diverse and high-quality dataset which represents static real-world HTML/CSS webpages. The dataset should (1) include multiple domains and layouts, (2) contain annotated accessibility violations and (3) has 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.

3.2 Construction

Two promising examples in the field of Image-to-Code are *Design2Code* [Si+24] and *Webcode2m* [Gui+24]. Both have used existing, large datasets and applied different processing steps to filter bad examples and remove noise or redundancy from the code. Based on their dataset curation, both serve as a good base for this thesis.
Therefore, we decided to use both datasets and manually select 53 high-quality data entries. Those 53 data entries consist of 28 entries from *Design2Code* and 25 entries from *Webcode2m*. In order to compare them on a fair basis, we only collect webpages that have english as their primary language.

3.2.1 Content Distribution

By using data entries of various domains and different layouts, we make sure to get a fair representation of the distribution of webpages in the real world. Based on our manual selection, we present the domain distribution in a pie chart in Figure 1.

3.3 Dataset Alignment

Due to the fact that Design2Code and Webcode2m use different strategies to purify their data, it is necessary to align both datasets. This includes (1) removing all external dependencies such as multimedia files (images, audio, videos, links), replacing them with placeholders such as src="placeholder.jpg" for images or href="#" for <a> Tags. Furthermore, (2) scripts and other dynamic contents are neutralised. Lastly, (3) non-visible content, such as advertisement-related tags or hidden elements are removed, because they are not required in an Image-to-Code environment and could possibly bias the accessibility score negatively.

3.4 Data Leakage

In a last step we try to rule out the risk of data leakage. Both datasets have been published on Huggingface within three months of the official knowledge-cutoff of *GPT-4o* and *Gemini flash 2.0*. This means that theoretically both datasets could have been part of the LLMs' training data.

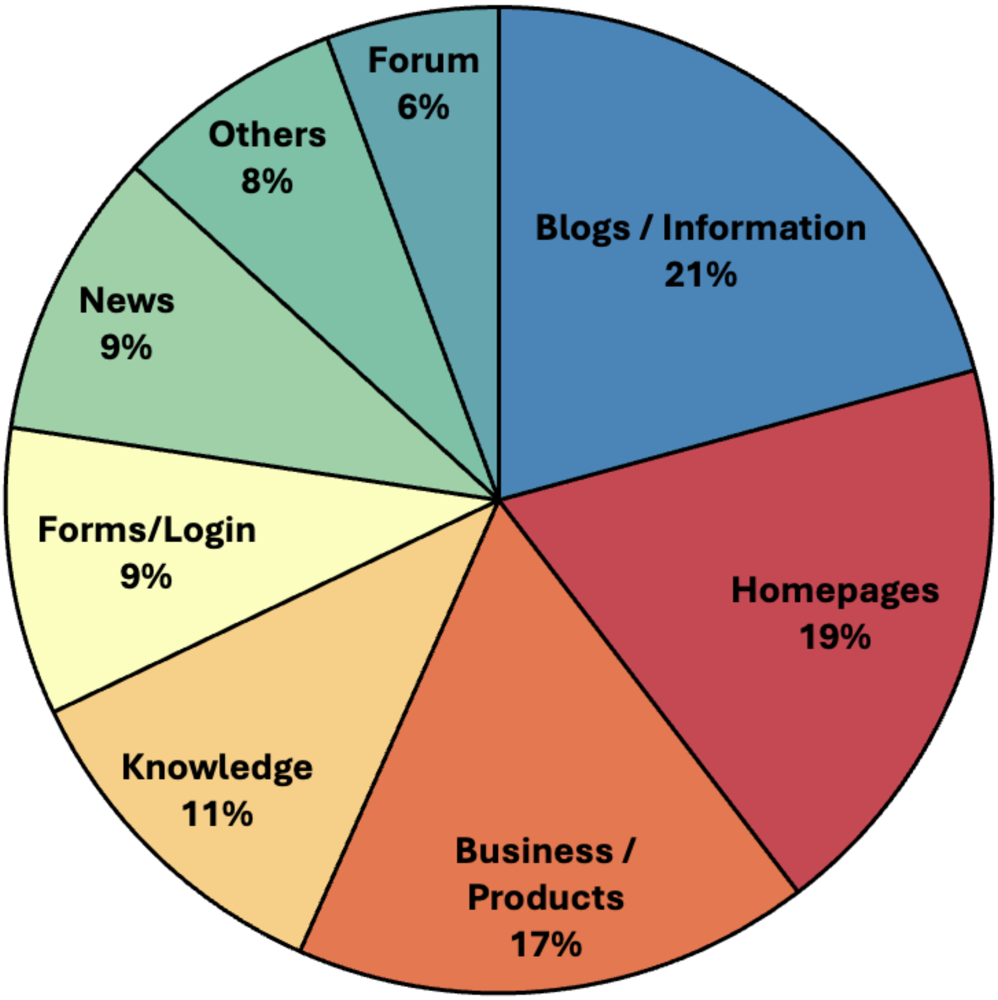


Figure 3.1: Distribution of Topics in Dataset

Therefore, we create a new, leakage-proof dataset of 20 entries which we test the LLMs on and compare their performance. This test dataset has 20 data entries and we construct it as follows:

- **Mutation of existing Data Entries:** We use 10 randomly-selected data entries of our existing dataset. Each data entry has been (1) rewritten by an LLM. While the meaning and the length (max ±20 %) remains roughly the same, the wording changes completely. The text appearance (2) is further changed by a set of 5 commonly used fonts in webpages and by changing (3) the *HUE* color code of each element slightly based on a random shift (±20 degrees).
  Additionally, we randomly change (4) the structure of the data entries manually. Structural blocks, such as headers, tables and images, are shuffled within and across different pages. This ensures a completely new layout, while making sure that the new data entries remain realistic and similar to their real-world origin.

- **Collection of new Data Entries:** The other 10 data entries have been crawled from two Github repositories created in 2025. *AlphaOneLabs* [Lab25] is an educational project with different webpage styles and layouts. The second repository *E-Commerce-Site* [Fer25] offers a wide range of e-commerce related webpages. On this basis, we randomly sample 10 diverse webpages.

3.4.1 Results

Across all prompting techniques, *GPT-4o* and *Gemini flash 2.0* performed better on the *leakage-test split* than on the original dataset in terms of their final scores. Since the results do not show any statistically significant drop in performance, we do not have any evidence for data leakage and assume that the LLMs have not seen the existing data. We proceed with the full dataset in the following experiments.

4 Benchmarks

4.1 Visual and Structural Similarity

The main instruction for LLMs for this experiment remains an Image-to-Code task, where we input a webpage screenshot and expect a working HTML/CSS as output. Therefore, it is necessary to evaluate the generated HTML/CSS code based on visual and structural similarity. One fine-grained approach has been introduced by *Design2Code* [Si+24]. It compares the input against the output based on component-level metrics rather than in its entirety. The matching algorithm combines the HTML elements in the ground truth with those in the generated code. Based on this matching, the following metrics are calculated and combined with equal weights in a final score:

- **Text-similarity:** Compares the text content of the HTML elements based on the sequence-matching algorithm of Python's difflib library.
- **Position-similarity:** Evaluates the position fidelity of the HTML elements by comparing the bounding boxes of the elements in the ground truth and the generated code.
- **Color-difference:** Calculates the text color difference between two blocks using the *CIEDE2000* color difference formula.
- **Clip score:** Uses the *CLIP* model to compare the visual similarity.
- **Area sum score:** Calculates the area of the bounding boxes and compares its size.

The combination of those scores allows to get a balanced view of the visual and structural similarity of the input and output.

4.2 Accessibility Metrics

Apart from the amount of accessibility violations, we measure and compare the accessibility improvements in terms of quantity and severity with two metrics:

- The *Inaccessibility Rate* (IR) has been used in previous, comparable research [AAM20]. It measures the percentage of DOM nodes with violations compared to the total amount of DOM nodes. Therefore, it divides the amount of nodes with accessibility violations by the amount of nodes which are susceptible for violations.

$$\mathrm{IR} = \frac{N_{\text{violations}}}{N_{\text{total}}} \tag{4.1}$$

- To capture the severity of accessibility violations according to the WCAG impact levels, we introduce the *Impact-Weighted Inaccessibility Rate* (IWIR). It uses the pre-defined impacts of accessibility violations found (minor, moderate, serious, critical) and assigns them to a value (1, 3, 6, 10). This scoring reflects the non-linear increase in impact for people with disablities, if a violation with a higher impact takes place within the code. Finally, we normalise the sum of all impact values by the worst-case scenario.

$$\mathrm{IWIR} = \frac{\sum_{i=1}^{k} v_i \, w_i}{\sum_{i=1}^{k} v_i \, w_{\max}} \tag{4.2}$$

Table 4.1: Severity weights used in IWIR.

Impact level	WCAG level	Weight w_i
Minor	AAA	1
Moderate	AA or AAA	3
Serious	A or AA	6
Critical	A	10

The combination of both metrics allows us to understand wether LLMs can not only decrease the amount of accessibility violations, but also its severity.

4.2.1 Accessibility Tools

The accessibility compliance has become a central issue for developers. Nowadays, there exist many accessibility tools which are specialized in detecting accessibility violations. According to former studies, automated testing can detect up to ∼60% of accessibility issues [Deq21]. This makes them valuable for developers, however they can not replace manual testing completely. However a combination of various tools can help to minimize the oversight of accessibility violations during the tests. Therefore, we decided to use combine 3 light-weight but complementary tools for our experiment. They work in similar ways, however they differentiate in their approaches.

- **Axe-Core (4.10.3):** The Axe-Core engine uses a *zero false-positives* approach. This means that the engine highlights only those violations which are certain to be violations. This is a conservative approach which can possibly lead to *false-negatives* in some cases.
- **Google Lighthouse Accessibility (12.4.0):** Even if Google Lighthouse Accessibility works with the same axe-core engine, it found additional, complementary violations during tests.
- **Pa11y (8.0.0):** During tests on our dataset, we found out that pa11y has strengths in detecting violations, especially in areas where the other tools seem to fail. Therefore, we decide to use it as our third complementary tool.

While this combination of different tools might not clear false-negatives, it can help to minimize and reduce their occurrence.

Cross-Tool Mapping

Generally WCAG standards are based on a predefined multi-level structure. The *principle* (perceivable, operable, understandable, robust) builds the first level and defines the main categories of accessibility. The second level - the *guidelines* are based on the principles. The third level are the *success criteria* which are based on the guidelines. The success criteria are the actual accessibility standards which can be checked. The last level - *the techniques* are used to check the success criteria. They are more fine-grained than success criteria and split into multiple different checks.

While the operating principles of axe-core, lighthouse and pa11y are similar, their output varies since they operate on different reporting levels. While axe-core and lighthouse operate on the success criterion level, pa11y operates on the level of techniques. In order to aggregate the three tools and de-duplicate the violations found, we create a JSON mapping covering the ∼90 most common WCAG techniques and ∼55 success criteria. With the help of this mapping, the output of all three tools can be combined in an automatic pipeline.

5 Experiment Design

5.1 Experiment Overview

This study tests the capabilities of modern vision language models (VLMs) to see how well they can move beyond pixel faithfulnuss and automatically produce accessible front-end code. Figure xy demonstrates our automatic Image-to-Code pipeline where an image with instructions is given as input and standards-compliant HTML/CSS is generated by the models. The output is then analyzed within multiple stages to evaluate visual and structural similarity, as well as *WCAG 2.2* compliance.
In order to account for the probabilistic nature of LLMs, each experiment will be repeated within 3 experiment runs.

5.1.1 Model Selection

The selection of the LLMs is a decisive factor for the interpretation of the results. To see the diffences of performance, we decided to use different types of state-of-the-art models. All have native vision capabilities, but they differentiate in size, architecture and target audience. In order to provide a general picture, we identified two model groups of interest:

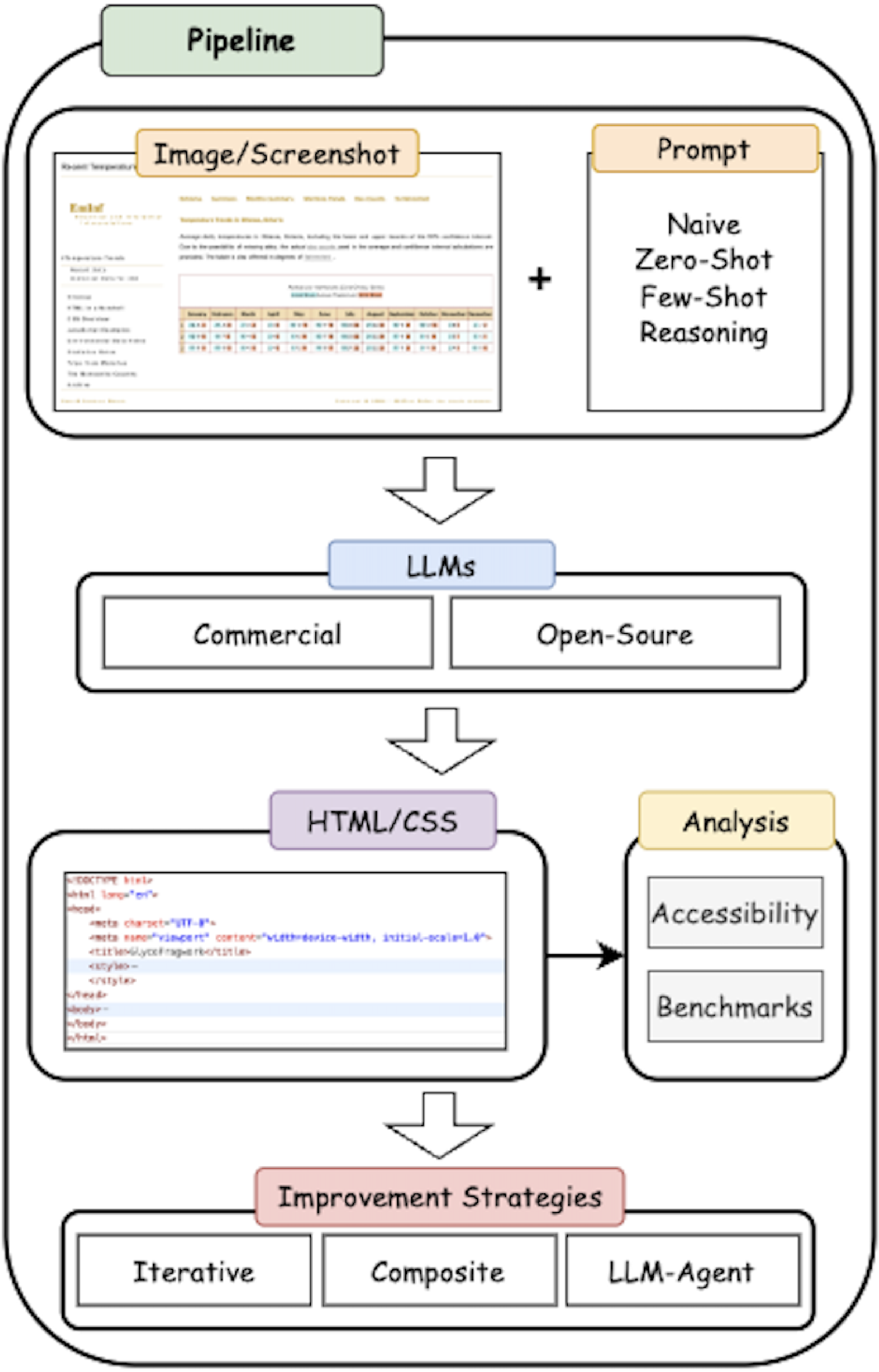
- **Commercial:** Big, commercial models build the foundation of our tests. We use *ChatGPT-4o* and *Gemini Flash 2.0* as representatives of this group. They have not been specifically trained for Image-to-Code tasks, but prior papers already show promising results in this field.
- **Open-Source:** Small, open-source models build the second group. While they are signifant smaller than the commercial models, theoretically they could be hosted by anyone and therefore might be more accessible for the public. The representatives of this group are *Llava 7B* and *Qwen 7B vl*.

5.2 Prompting Techniques

To understand the models' capabilities to generate accessible code, we test them with different prompting techniques (full text in appendix). Each prompt provides the screenshot (B64 decoded) and a description of the task.
Prior research has shown that more advanced prompting techniques can help to improve the generation of robust and accurate code. The wording of the prompts is similar to the one of prior research [Suh+25; Xia+24]. The content is the same while we have enriched the prompts with more details.

5.2.1 Naive

The naive approach only instructs the LLMs to accurately fulfill Image-to-Code tasks without mentioning accessibility in its prompt. This shows how state-of-the-art LLMs perform in generating accessible code naturally without instructing it to do so.



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>[illegible]</title>
    <style>
    </style>
</head>
<body>
</body>
</html>
```

Figure 5.1: Experiment Pipeline

5.2.2 Zero-Shot

The zero-shot approach resembles the naive approach in terms of the Image-to-Code instructions. However, here the LLMs are explicitly instructed to obey the WCAG 2.2 standards. The prompt does not contain examples, however it emphasizes accessibility by reminding it about the WCAG standards including a link to official WCAG standards.

5.2.3 Few-Shot

The few-shot approach resembles the zero-shot prompt, however it is enriched with explicit examples to support the LLM to generate more accessible code. Since the number of possible examples is too large, we decided to provide examples for only 8 of the most common accessibility violations (e.g. Missing Alt-Text, Color Contrast). For this approach, we have followed the structure of previous works [Suh+25]. The structure of the examples contains the rule's name, a description, a correct example and a wrong counter example. If possible, the examples were taken from the official W3C website [24].

5.2.4 Chain-of-Thought

The chain-of-thought prompt is used to let the LLMs reason about the task and possible accessibility problems within the generation. Similar to prior work, we use the reasoning instructions Ët's think step by step.änd instruct the model to output its reasoning comments which are then stripped in a pre-processing step. [Cha+24].

5.3 Improvement Strategies

To solve the limitations of current prompting techniques, we test three advanced improvement strategies that explicitly correct accessibility violations. One is an iterative approach that uses the feedback from accessibility tools in an iterative manner to improve the code. The second is a composite approach which combines pre-processing and post-processing techniques to improve the LLMs' output. The last approach is a multi-agent approach that uses further state-of-the-art LLMs to detect and solve accessibility violations.
The advanced strategies are compared to old approaches and the capabilities of AI models to resolve accessibility violations are analyzed.

5.3.1 Iterative Self-Critique

The iterative approach follows a similar approach to recent findings in the area of generating more accessible code [Suh+25]. Starting with the naive generation, we run axe-core, lighthouse and pa11y. Afterwards, the accessibility violations found are incorporated with a refinement message to the LLMs. The objective is to instruct the LLMs to create a more accessible code based on the violations found in the code. This feedback is given to the models up to three times. If the LLMs generate code without any accessibility violations within one round, the iterations stop.

5.3.2 Color-Aware Feedback

Using the block detection algorithm of *Design2Code* [Si+24], we calculate the bounding boxes of each UI text component in an image, without using *OCR* (Optical Character Recognition). With the help of the bounding boxes, we determine the font and background color of each

UI component. Based on those colors, the relative luminance and the color contrast ratio is calculated. If a color contrast ratio fails to comply with the WCAG color contrast requirements, a new color that fullfills this guideline is automatically recommended and injected into a refinement prompt.
Apart from the recommended color, the refinement prompt contains also other accessibility violations found, similar to the iterative approach. In contrast to the iterative approach, we only use one self-critique round.

5.3.3 Iterative Multi-Agent

In a combination with different agents, this approach uses a 3-layer architecture to detect, identify and fix violations in the code. In comparison to the approaches above, it is fully based on LLMs and does not use any pre- or post-processing techniques.
The first layer, called *Issue Detector*, finds the place in the code which violates a certain standard and outputs the violations in a pre-defined format. Based on this output, the second layer *Issue Identifier* classifies the issues. It identifies them based on the WCAG 2.2 standards by adding the guideline's name and the severity of the violation. In a last layer, the *Issue Resolver* is responsible to path the violations. In order to be as precise as possible, without adding new violations, we resolve the violations in batch sizes of *n=5*.
This architecture allows a clean separation between the location, the type and the solution of accessibility violations.

6 Evaluation

6.1 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.

6.1.1 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 ∼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 ∼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.

6.1.2 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 <div> 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

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

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 [Web25] 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 <div> 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 63% of all misses, 69% if black (#000000) and white (#ffffff) are included.
- **GPT-4o.** 11 out of 237 colors match, leading to 23% 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.

6.2 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

Table 6.1: Most common colors used by the LLMs.

Model	Distinct Colors with Violations	Top-7 colors	Percentage of all color contrast violations
GPT-4o	127	#ffffff, #777777, #007bff, #0000ee, #888888, #999999, #29abe2	tbd
Gemini Flash 2.0	237	#ffffff, #777777, #888888, #999999, #ff0000, #007bff, #808080	tbd
Qwen 2.5-vl	0		tbd

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.

7 Conclusion

7.1 Section

ncoh bessere Überschrift finden

7.1.1 Future Directions

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

8 Appendix

8.1 Results Data Leakage

Table 8.1: 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 8.2: 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

8.2 Prompts

Naive Prompt

(Instruction):

You are an expert web developer who specializes in HTML/CSS. Your task is to replicate the provided UI screenshot of a webpage pixel-perfectly using HTML/CSS.

[Guidelines]:

1. **Layout: Structure the markup so the spatial arrangement of every element exactly matches the screenshot.**
2. **Styling: Reproduce fonts, colors, spacing, sizes, borders, shadows, and any other visual details as closely as possible.**
3. **Content: Include all visible text, icons, and graphic elements.**
4. **Image placeholders: Blue boxes represent images. Use '<img>' and as a source the placeholder file 'src="src/placeholder.jpg"' to reserve space. However, the appropriate height and width is very important.**
5. **Delivery format: Output only the complete HTML and CSS code in one file — no additional comments or explanations.**

Now convert the following image into HTML/CSS according to these requirements.

Accessibility Reminder

(Instruction):

It is EXTREMELY important that your HTML/CSS complies with WCAG 2.2.

Refer to the full spec if in doubt:

`Link to WCAG 2.2`

Avoid any violations.

Zero-Shot Prompt

(Instruction):

`Naive Prompt`

+

`Accessibility Reminder`

Few-Shot Prompt

(Instruction):

`Naive Prompt`

+

`Accessibility Reminder`

+

Detailed accessibility rules with correct/incorrect examples for each rule. Each rule comes with [name, description, correct_example, counter_example].

+

8 Few-Shot examples.

Chain-of-Thought Prompt (CoT)

(Instruction):

`Naive Prompt`

+

`Accessibility Reminder`

+

Let's think step by step.

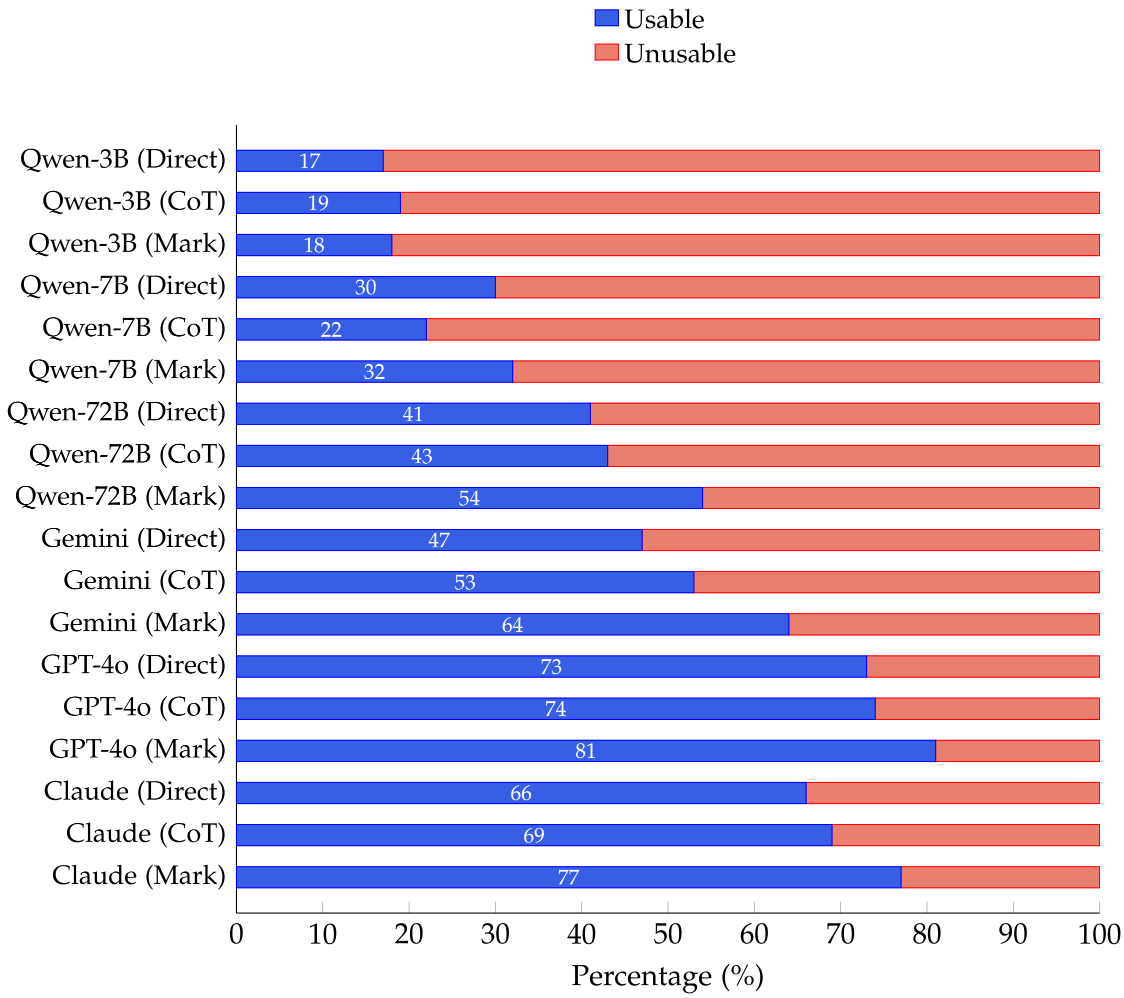
Output Format

When you reply:

1. **Return ONE valid JSON object and nothing else.**
2. **It MUST have exactly these keys:**
   - **"thoughts" – your internal reasoning**
   - **"code" – a complete HTML/CSS document as described above**

Figure 8.1: Overview of Prompts.

8.3 Accessibility Violations Resolved



List of Figures

List of Tables

Bibliography

[24] *Web Content Accessibility Guidelines (WCAG) 2.2*. W3C Recommendation. World Wide Web Consortium, Dec. 12, 2024.

[AAM20] 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.

[Alj+24] 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.

[Bel17] 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.

[Cha+24] H. Chae, Y. Kim, S. Kim, K. T.-i. Ong, B.-w. Kwak, M. Kim, S. Kim, T. Kwon, J. Chung, Y. Yu, and J. Yeo. "Language Models as Compilers: Simulating Pseudocode Execution Improves Algorithmic Reasoning in Language Models." In: *arXiv preprint arXiv:2404.02575* (2024). DOI: 10.48550/arXiv.2404.02575. arXiv: 2404.02575.

[Deq21] 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.

[Fer25] N. Ferhan. *E-Commerce Site*. https://github.com/nuranferhan/e-commerce-site. GitHub repository. 2025.

[Gui+24] 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.

[Lab25] A. Labs. *AlphaOneLabs Education Website*. https://github.com/alphaonelabs/alphaonelabs-education-website. GitHub repository. 2025.

[Si+24] 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.

[Suh+25] 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.

[Wan+24] 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.

[Web25] WebAIM. *The WebAIM Million - An annual accessibility analysis of the top 1,000,000 home pages*. Accessed: 2025-06-16. 2025.

[Xia+24] 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`.