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 Section

High quality web user interfaces are the backbone of our modern society. They allow us to present products or services in an interactive way and reach billions of users every day. However, the creation of Websites or user interfaces (UIs) follows a similar and repetitive pattern.

First UI designs are created with the help of special design tools. The UI designs present the foundation for software developers. In a second step, those designs are translated into functional UI code which tries to resemble the intended layout and structure, but also obey to other design aspects.

One essential, but yet frequently underestimated aspect in this process is *accessibility*: According to the official *WCAG* guidelines, code must be perceivable, operable, understandable and robust for people with various disabilities.

Complying with accessibility standards is not only an optional, moral aspect of web development, but it now has to follow regulatory boundaries. Ensuring accessibility is no longer optional. For instance the *European Accessibility Act* comes into effect on June 28, 2025 and obliges any e-commerce or digital service in the EU to comply with those standards.

Current Large-Language Models (LLMs) have shown signifant improvements in automatic code generation. Especially *Image-to-Code* tasks where UI designs are given as input and LLMs output the functional UI code, have demonstrated that especially for less complex webpages LLMs show good performance [Si+24]. Several benchmarks have shown the competitive performance of LLMs on those tasks. However, the capability of modern LLMs to generate accessible Code in an Image-to-Code environment has only started to gain researchers interest quite recently. Existing research in this field have compared human to the generated code and tried to better align with the accessibility standards. Nevertheless, this has never been tested in an Image-to-Code environment. Apart from that, accessibility does not only concern the visual appearance of a web user interace, but also its functionality. Thus, it requires the LLMs to have a deeper understanding than in classical Image-to-Code scenarios where LLMs only reproduce the input images pixel perfectly.

1.1.1 Our Contributions

In order to close this gap, we propose a large scale accessibility evaluation pipeline of LLM-based Image-to-Code generation. Even if the main contribution of this thesis is in the field of accessibility, the visual and structural similarity will also be taken into account.

For this pipeline we use a dataset as input which contains of 53 real-world webpage examples which have been gathered from existing datasets and mutated in order to minimize data leakage. It covers a wide spectrum of layouts, content areas and accessibility features. This dataset is the ground truth and also contains information about the accessibility violations of the human developers.

This dataset is then used in a reproducible evaluation pipeline which measures both the visual and structural similarity (pixel/DOM fidelity), as well as the accessibility compliance.

Therefore, we propose different benchmarks in order to measure the performance of the LLMs. The corresponding data for the benchmarks will be evaluated during analyses after the code generation.
This environment will be tested in an in-depth comparison of 4 state-of-the-art LLMs with vision capabilities (gpt-4o, gemini flash 2.0, llama 3.2 vision, qwen 7B vl). Those LLMs are tested across the images of the dataset and different prompting strategies. We also propose a technique and implementation details to get the best results.

2 Related Work

2.1 Background

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

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. As *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. A prominent example is *DCGen* [Wan+24] where researchers have segmented screenshots into smaller, visual segments, let LLMs generate code for each segment and reassemble them 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 found across one million webpages tested [Web25]. Since accessibility compliance does not only improve the user experience for people with disabilities, but also helps with *Search Engine Optimization* (SEO), it is not surprising to see recent research in this area. 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. Especially due to rising frontend code generation, it will be interesting to see how the WCAG compliance will influence the visual similarity.

2.4 AI-enhanced GUI testing

if necessary...

3 Dataset

3.1 Construction

The main goal is to gather a diverse and high-quality dataset which represents static real-world HTML/CSS webpages. This includes different layouts, components and contents. In the past, there have been different attempts to collect a dataset fullfilling exactly those requirements.
Two promising examples in this field are *Design2Code* and *Webcode2m*. 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.

3.1.1 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.

3.2 Dataset Mutation

Due to the fact that Design2Code and Webcode2m use different strategies to purify their data, it is necessary to align both datasets in order to get a fair comparison. This includes removing all external dependencies such as multimedia files (e.g. images , audio, videos, . . .) from the datasets. Furhtermore, this means adding placeholders like src=p̈laceholder.jpgf̈or images or href=#̈f̈or <a> Tags. Lastly, we remove all of the non-visible content (advertisement-related, hidden) of the webpages, because it is not necessary in an Image-to-Code environment and could only add negatively to the accessibility score.

3.3 Data Leakage

In a last step we try to rule out the risk of data leakage. Both datasets have been uploaded to Huggingface a few months before the official knowledge-cutoff of some of the LLMs, which we use, and theoretically, they were publicly available at that time. While Design2Code has uploaded its data 3 months before the knowledge-cutoff, in the case of Webcode2m, it was only 2 months. To tackle this issue, we create a new, leakage-proof dataset of 20 entries which we input to the LLMs and compare their performance. If the performance in terms of our Benchmarks is comparable to our the one of our existing dataset, we assume that no data leakage has occurred.
This test dataset has 20 data entries and has been collected the following way:

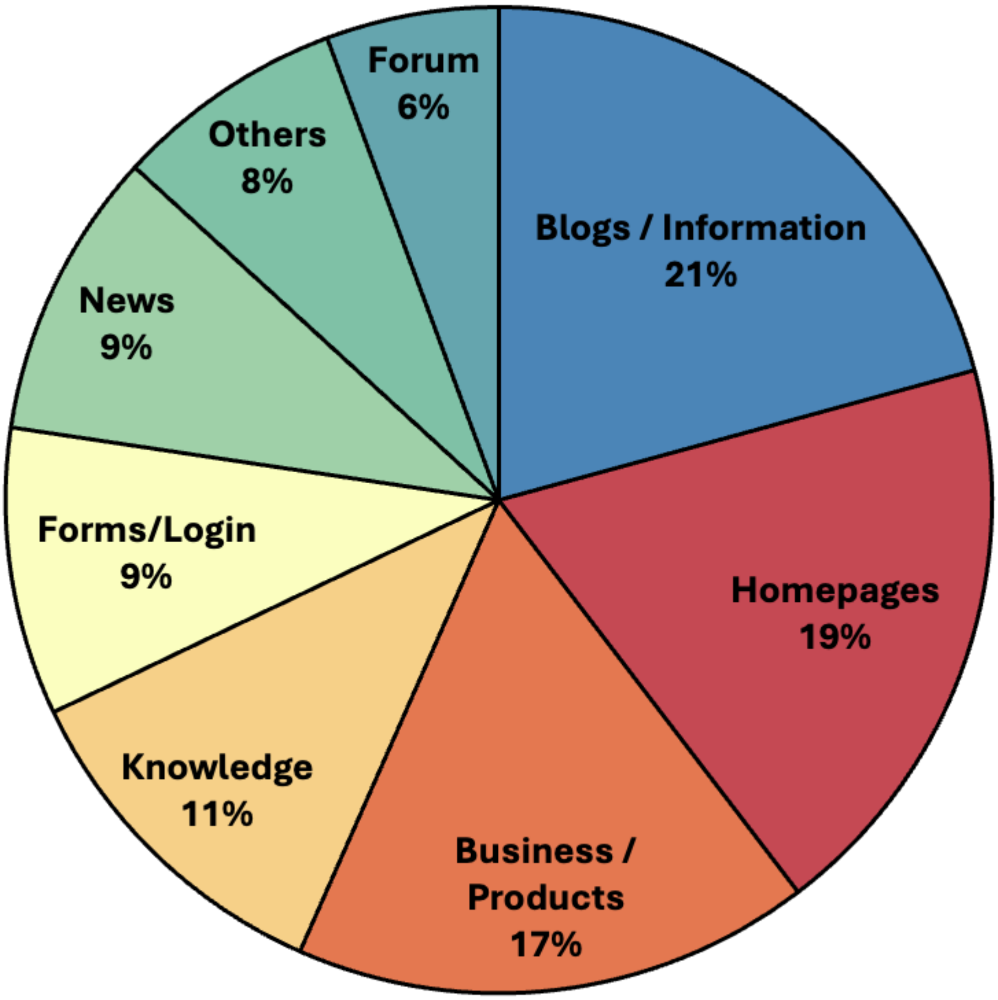


Figure 3.1: Distribution of Topics in Dataset

- **Mutation of existing Data Entries:** We use 10 randomly-selected data entries of our existing dataset.
- **Collection of new Data Entries:** The other 10 data entries have been collected from two Github repositories which contain frontend webpages. Those webpages have been crawled and added to the dataset.

3.3.1 Mutation of existing Data Entries

The mutation of the data entries tries to change the existing ones without losing the reference to the real-world. The goal is to change the data entries in such a way, that the LLMs cannot memorize the data entries, without adding artificial noise or new accessibility violations. Therefore, we apply the following mutations to the data entries:

- **Text:** The entire text has been rewritten by a LLM. While the meaning and the length (max ±20 %) remains roughly the same, the wording changes completely, in order to avoid memorization based on text snippets.
- **Text Font:** To change the visual appearance of the data entries, we define a set of 5 commonly used fonts in webpages. Based on a random sample, we change the text font for each data entry.
- **Colors:** The *HUE* color code of each element is slightly changed based on a random shift (±20 degrees). Apart from that, the saturation and lightness of the color is changed by a maximum of ±20 %.

Since those changes remain quite superficial and might not be enough to rule out data leakage, we further alter those data entries. We randomly change the structure of the data entries manually. Components, such as tables, images and text are interchanged within one html/css file in order to alter the webpage's layout. Apart from that, we implement cross-web exchanges, similar to former research (paper). We randomly select an order and exchange header and footer within different files. This ensures a completely new layout, while making sure that the new data entries remain realistic and similar to their real-world parents.

3.3.2 Collection of new Data

In a second step we collect webpages of two Github repositories which have been created in 2025. The first repository (source) is an educational project with many different webpage styles and layouts. The second repository (source) offers a wide range of e-commerce related webpages.
We crawl their html/css, pay attention on diverse content and randomly select 10 examples of this pool.

3.3.3 Results

As can be seen in the final results in the appendix, across all different prompting techniques both models performed better on the *Data Leakage Test Dataset* in terms of the combined final score. The results also do not show a statistical signifcant difference in terms of the benchmarks between both datasets. Since we do not have any evidence for data leakage, we assume that the LLMs have not seen the existing data and therefore continue with the tests.

4 Benchmarks

4.1 Visual and Structural

As the main instruction for LLMs for this study remains an Image-to-Code task, it is necessary to evaluate the generated HTML/CSS code based on visual and structural similarity. One approach which seems very promising has been presented in the paper Design2Code. This approach is more fine-grained than former ideas, since it compares the input and the output on a component-level rather than in its entirety. The authors describe a sophisticated matching algorithm that combines the HTML elements in the ground truth with those in the generated code. Based on this matching, it is then possible to run several metrics, such as text-similarity, position-similarity, color-difference, clip score and area sum score.

4.2 Accessibility

In order to measure and compare the accessibility improvements in terms of quantity and severity of the violations we use two different metrics.
The first metric is the *Inaccessibility Rate* (IR) which has been used in previous research in this field [AAM20]. This metric divides the amount of nodes with accessibility violations by the amount of nodes which are susceptible for violations. The interpretation of this metric is straight-forward, since it allows us to get the percentage of nodes with violations compared to the total amount of nodes.

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

However, the Inaccessibility Rate does not take the severity of issues into account. Thus, we have created the *Impact-Weighted Inaccessibility Rate* (IWIR). This metric uses the impacts of Accessibility Violations found (minor, moderate, serious, critical) and assigns them to a value (1, 3, 6, 10). Our 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, the Impact-Weighted Inaccessibility Rate is calculated by creating the sum over all Violations found multiplied by the corresponding value for the severity. This sum is then divided by the amount of violations found multiplied by the highest impact score. By dividing the sum above through the worst possible outcome, a situation where every violation is critical, allows to get an understanding of the severity of the violations found.

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

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

As prior papers have estimated, $\approx 96\%$ contain accessibility violations in its source code. Thus, the accessibility compliance remains a central issue for developers (paper). Nowadays, there exist many accessibility tools which are specialized in detecting violations. They work in similar ways, however they differentiate in their approaches. It is important to mention that there still exist accessibility standards which can only be checked with manual tests. No tool can analyze all violations automatically, however a combination of various tools in this area can help to minimize the oversight of accessibility violations during the tests. Therefore for our experiment, we decided to use combine 3 light-weight but high-precision tools in this field.
The Axe-Core engine is a promising candidate in this fields, due to its *zero false-positives* approach. This means that this engine works more conservative in terms of reporting accessibility violations than other tools. We use the eponymous tool axe-core which is based on this engine. Google Lighthouse Accessibility works with the same engine, however during tests, it found additional violations which made it another promising candidate.
Lastly during test on our dataset, we found out that in for accessibility standards axe-core seems to be too conservative and produces *false-negatives*. While they can not be extinguished completely, they can be reduced by combining it with another tool. Based on those results, *pa11y* completes the list of automatic accessibility tools used in this pipeline. Especially in areas where the other tools seem to fail reporting violations, it has its strengths.
While this combination of different tools might not clear false-negatives, it can help to minimize their occurrence.

Mapping between Tools

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 are the guidelines which 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 are the techniques which 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. This is because they operate on different reporting levels. While axe-core and lighthouse operate on the level of success criterion, pa11y operates on the level of techniques. In order to combine the three tools and detect similar violations, this inequality requires a manual mapping of techniques to their corresponding success criterion. This has been implemented for the most common 200 pa11y violations and 100 axe-core (Zahlen?) violations. This mapping is used to combine the results of the three tools.

5 Experiment

5.1 Experiment Setup

In this thesis, we test the capabilities of LLMs to generate accessible code in an Image-to-Code environment. Therefore, a pipeline is used which provides the LLMs an image with instructions to replicate this image in HTML/CSS. In the following, advanced prompting techniques are also provided to guide the LLMs to generate accessible code. The output of the LLMs for each parameter tuple (Model, Prompting Technique) is then analyzed on visual and structural similarity, as well as on accessibility compliance, based on WCAG 2.2 standards. In order to cope with the probabilistic nature of LLMs, each experiment with a predefined tuple of parameters will be tested within 3 experiment runs.
After the generation of the LLMs' output for the Image-to-Code task, the HTML/CSS is first analyzed on visual and structural similarity as described above and afterwards analyzed on accessibility violations with the help of axe-core, lighthouse and pa11y.

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. They differentiate in size, architecture and use case, but they all have vision capabilities in common. In order to provide a general picture, we identified three model groups of interest:

- **Commercial:** Big, commercial models build the foundation of our tests. We use *gpt-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.

- **Small, Open-Source:** Small, open-source models build the second group. Theoretically, they should be more accessible for the public and could be hosted on local machines. The representatives of this group are *llama 3.2 vision* and *qwen 7B vl*. While the models are signifant smaller, they still have been trained on a lot of data.

- **Fine-tuned:** The last group are fine-tuned models. In this case, we consider models which have been trained specifically on similar Image-to-Code tasks. We use *VLM WebSight finetuned* which is a fine-tuned version official the models *SigLIP* and *Mistral 7B v0.1*. It has been trained on the *Websight* dataset which contains more than 800,000 HTML/CSS data entries.

5.2 Prompting Techniques

In order to understand the models' capabilities to generate accessible code, we test the models with different prompting techniques. Prior research has shown that more advanced prompting techniques can help to improve the generation of robust and accurate code. Therefore, we combine commonly used techniques with advanced prompting techniques to guide the LLMs

to create more accessible 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.
All prompts can be seen in the appendix.

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.

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. 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 Reasoning

The reasoning prompt is used to let the LLMs reason about the task and possible accessibility problems within the generation. Similar to prior work, we use a prompt to generate reasoning steps with L̈et's think step by step.äs part of the output [Cha+24].

5.2.5 Iterative

The iterative approach follows a similar approach to recent findings in the area of generating more accessible code [Suh+25]. This prompting technique lets the LLMs generate Image-to-Code tasks by using the naive prompting technique in the base iteration. Afterwards, the generated output is analyzed and 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. The iterative approach contains one naive prompt and three rounds of code refinement. If the LLMs generate code without any accessibility violations within one round, the iterations stop.

5.3 Advanced Accessibility Strategies

To solve the limitations of current prompting techniques, we test two advanced improvement strategies. One is a composite approach which combines pre-processing and post-processing techniques to improve the LLMs' output. The second 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 Color-Aware Iterative Approach

This approach uses an advanced block detection algorithm in order to receive the UI text components and its corresponding bounding boxes in an image. This algorithm is based on previous works [Si+24] and determines the components without using *OCR* (Optical Character Recognition). With the help of the bounding boxes, the font color and the background color of each UI component is determined. On this basis, the relative luminance and the color contrast ratio is calculated. If a color contrast ratio fails the requirements, a new color is recommended that fullfills this guideline.
After the generation of the frontend code by LLMs, similar to the iterative approach, the code is analyzed for accessibility violations. The violations found are combined with a refinement message to serve as input for a refinement round. In contrast to the iterative approach, we only use one round of improvements.

5.3.2 Multi-Agent Approach

In a combination with different agents, this approach uses a 3-layer architecture to detect, identify and fix violations in the code.
The first layer finds the place in the code which violates a certain standard. Only the marked regions are submitted to the next layer. The second layer identifies the issues. It classifies them based on the WCAG 2.2 standards by adding the name, its severity and an explanation of the violation. In a last layer, the violations are patched with another agent. It creates a solution and an explanation to the user.
This architecture allows a clean separation between the location, the type and the solution to accessibility violations. Its layout can be seen in image xxy.

6 Evaluation

6.1 Accessibility

In order to provide a comprehensive analysis of the accessibility violations, we divide the analysis into 3 categories. Those categories differentiate in terms of quantitive and qualitive analysis as well as in its depth.

6.1.1 Quantitive Analysis

Figure x shows the quantitive analysis of the amount and type of Accessibility Violations found during the 3 experiment runs for each tuple of parameters. Based on this figure, we can observe several interesting findings.
Firstly, the amount and types of Accessibility Violations show a similar patterns across the different models. While overall *gpt-4o* produced slightly less violations, its numbers and distribution show a similar structure. Without the refinement techniques, we also see a very skewed error profile. Color contrast violations, as well as HTML landmark and region issues are the predominantly cause for at least 75% of all violations in both gemini's flash-2.5 and in gpt-4o model and with diverse prompting techniques.
While gemini seems to have more color contrast violations, landmark and region rules cause more problems for the gpt-4o model. Those findings are confirmed across different parameters. The subsequent types of violations are in the fields of missing labels, links without distinguishable color, issues with the HTML header and the size of frontend components. The results show a noticeable trend that only a small subset of possible WCAG violations cause non-negligible amounts of violations in an Image-to-Code environment. Especially in the case of gpt-4o, there are only 6 types of violations which cause more than 10 violations combined across all prompting techniques in our dataset.
Similar to related papers in this field (paper xxx), our results also show that different prompting techniques only have small impact on the findings. 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 such as zero-shot or reasoning.
Figure xyy shows the violations found in the ground truth HTML/CSS of our dataset. 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. This shows that LLMs write by default more accessible code than humans by incorporating the most important rules. However, as described above they struggle with more complex rules such as color contrast or landmarks that require further information or thinking.

IR noch mit aufnehmen. IW-IR noch mit aufnehmen. Verteilung der Issues bei Human und LLms mit aufnehmen. Bar Charts noch mit aufnehmen ohne iterative

6.1.2 Qualitive Analysis

In a first step, we analyze how consistent violations are within different within different experiment runs and dataset entries. Figure xy shows the consistency of the type and the amount of a violation by using the *cosine-similarity*. Therefore, we analyze the type of violation and its amount per file in vector-like structure. The cosine-similarity is then calculated between the vectors of the different experiment runs and dataset entries. This allows us to analyze how similar the violations are and to understand how consistent the LLMs are in generating accessible code.

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.

6.2 Image-to-Code Similarity

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.

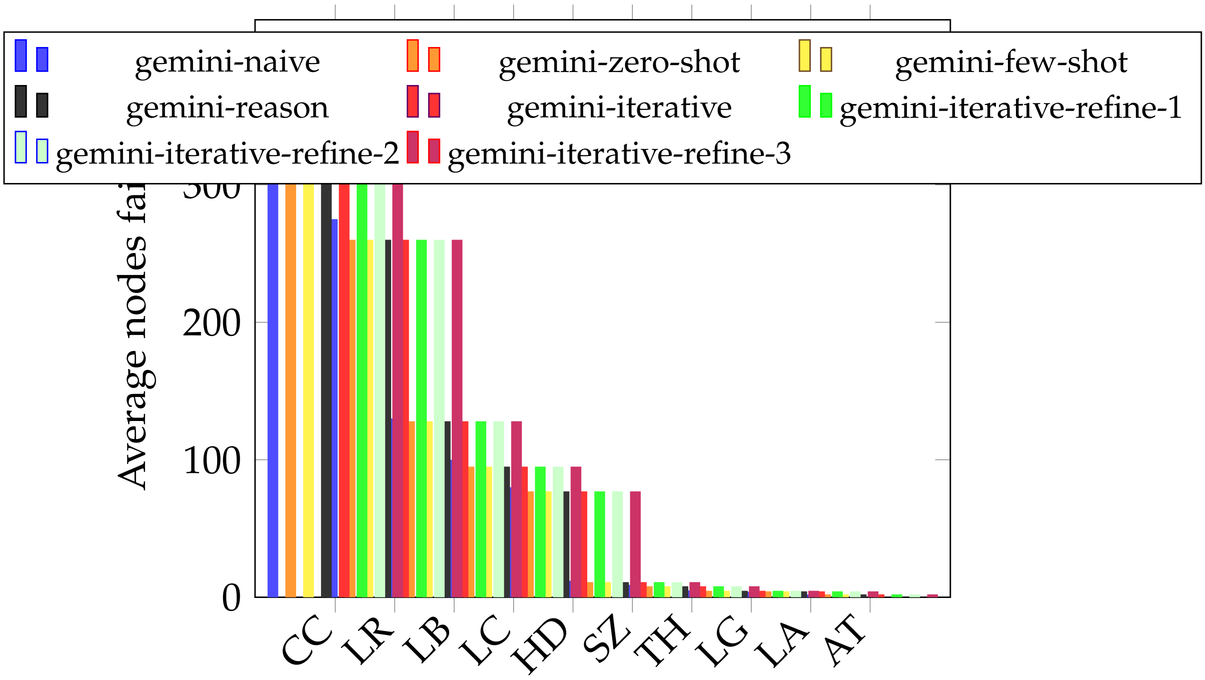


Figure 6.1: Mean number of WCAG violations per rule group. Abbreviations are explained in Table 6.1.

Table 6.1: Mapping of two-letter abbreviations to WCAG rule groups (used in Fig. 6.1).

Abbr.	WCAG rule group
CC	Color Contrast: Text
LR	Landmark & Region: Missing & Unique Landmarks
LB	Label: Multiple Elements; Content Missing
LC	Links: Distinguishable Color
HD	Headings: Wrong Order; Empty & Missing Headings
SZ	Size: Target Size; Element Too Small
TH	Tables: Table Headers
LG	Language: Document Lang Missing / Invalid
LA	Links: Missing Descriptive Content of <a>
AT	Alt-Text: Images

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**	0	0	0	0	0
	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**	0	0	0	0	0
	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**	0	0	0	0	0
	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**	0	0	0	0	0
	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.

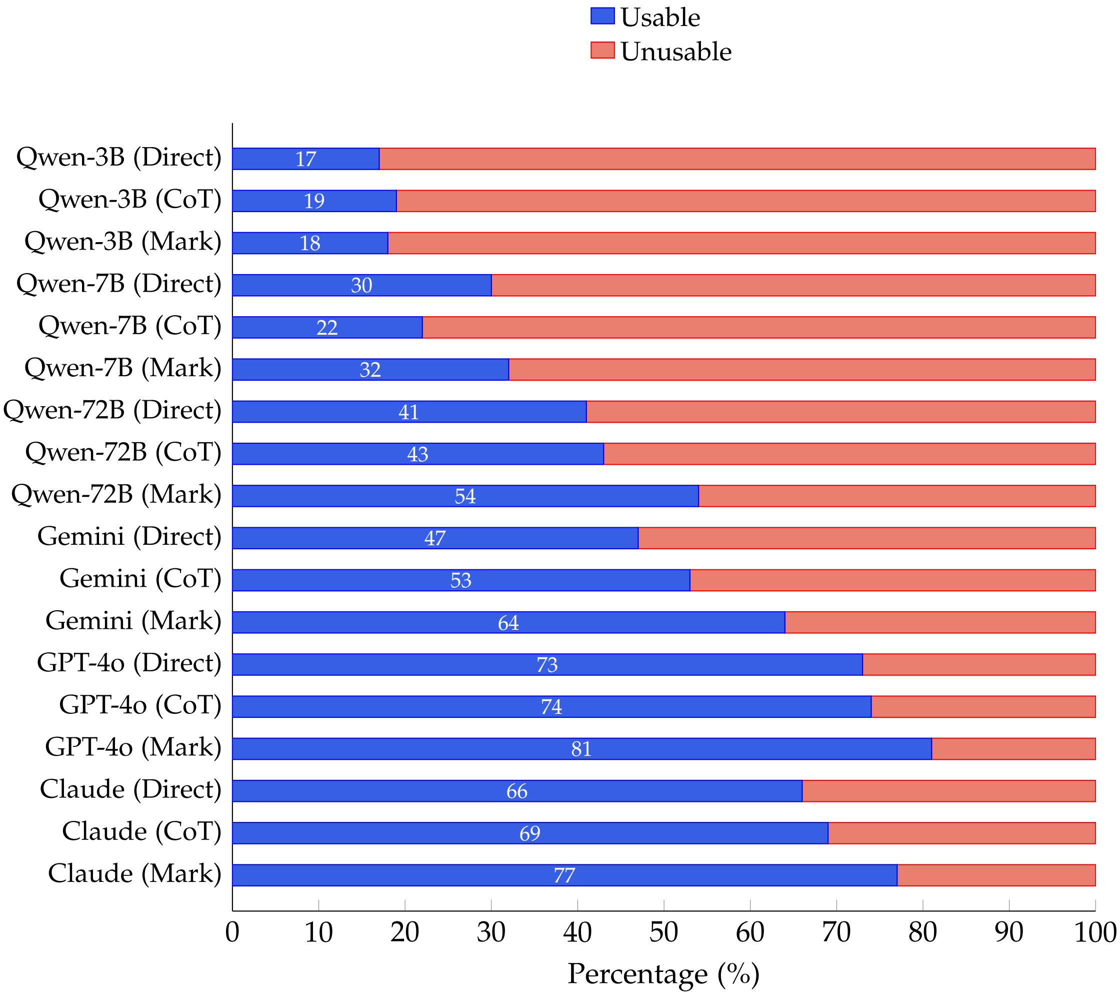
Output Format

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.

[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.