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 been tested by various researchers in the past. 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 Web Accessibility

asf

2.2 Image-to-Code

sided

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.

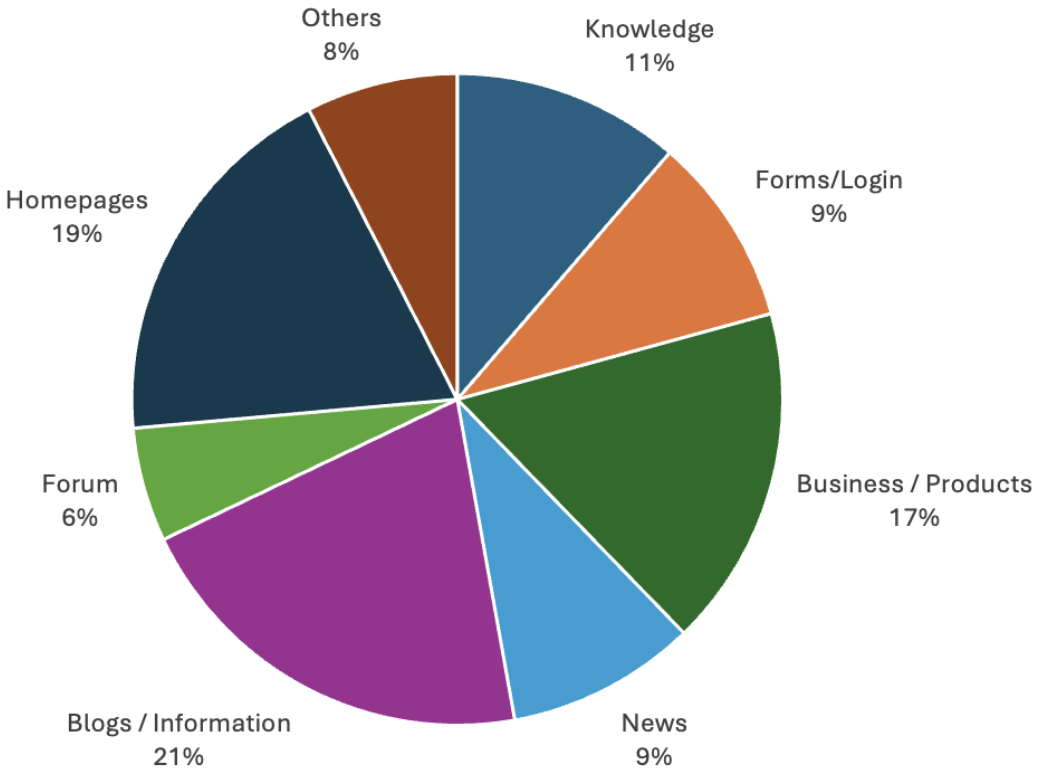

Figure 3.1: Distribution of Topics in Dataset

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:

- **Mutation of old Data:** We use 10 randomly-selected data entries of our existing dataset.
- **Collection of new Data:** 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 old Data

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

The final results can be seen in the appendix. However, they show no difference in terms of the benchmarks between the existing dataset and this new leakage-free dataset. 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 is then analyzed on visual and structural similarity, as well as on accessibility compliance, based on WCAG 2.2 standards.
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. 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 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.

5.2.4 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 internally with ̈Let's think step by step.̈before generating 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 Accessibility Evaluation

5.4 Similarity Evaluation

6 Accessibility

6.1 Section

6.1.1 Subsection

7 Conclusion

7.1 Section

7.1.1 Subsection

List of Figures

List of Tables

Bibliography

[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.

[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.

[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.