Advancing Code Generation from Visual Designs through Transformer-Based Architectures and Specialized Datasets

TOMMASO CALÒ, Politecnico Di Torino, Italy
LUIGI DE RUSSIS, Politecnico di Torino, Italy

Manually translating web designs into code is a costly and time-consuming process, particularly due to the frequent iterations and refinements between designers and developers. Deep learning techniques, which possess the capability to automatically translate designs into functional code using an encoder-decoder architecture, have emerged as a promising solution to enhance this tedious process. However, many current methods depend on simplistic datasets that do not capture the diversity of components found in modern websites. Additionally, the potential of transformer-based models, which have enabled significant progress in vision and language modeling tasks due to their scalability and ability to handle cross-modal relationships, has not been investigated in this context. Addressing these limitations, this paper contributes with: 1) a web scraping methodology to automatically collect and process a diverse dataset of real-world websites with reduced noise and complexity, 2) a synthetic dataset of webpage mockups along with their sketched conversions, and 3) an evaluation of two recent multimodal transformer architectures on these proposed datasets. Results on synthetic and sketch-based datasets demonstrate the architectures potential as effective design-to-code automation solutions, while identifying remaining challenges in modeling real-world website complexity.

CCS Concepts: • **Human-centered computing** → **User interface programming**; **Wireframes**.

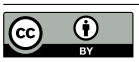
Additional Key Words and Phrases: Transformer-based models, Web design automation, Mockup-to-code, Sketch-to-code, Deep learning in UI design

ACM Reference Format:
Tommaso Calò and Luigi De Russis. 2025. Advancing Code Generation from Visual Designs through Transformer-Based Architectures and Specialized Datasets. *Proc. ACM Hum.-Comput. Interact.* 9, 4, Article EICS013 (June 2025), 37 pages. https://doi.org/10.1145/3734190

1 Introduction

As people's participation in the digital world continues to increase at a rapid pace, web applications have become crucial in establishing and maintaining an effective online presence for both individuals and organizations. The virtual face of an entity, whether it is a personal blog or a corporate site, plays an important role in creating the first impression and ensuring sustained engagement with the audience. As a result of their high importance, there is a growing requirement to develop complex, attractive, and sleek websites. A crucial step of website creation involves conceptualizing the User Interface (UI) design through various stages, from sketching to prototyping. This iterative process engages end users, designers, and developers, enabling them to deliberate upon the website's proposed layout, composition, and interactivity.

Authors' Contact Information: Tommaso Calò, Politecnico Di Torino, Dipartimento di Automatica e Informatica, Torino, Torino, Italy, tommaso.calo@polito.it; Luigi De Russis, Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy, luigi.derussis@polito.it.



ACM 2573-0142/2025/6-ARTEICS013
https://doi.org/10.1145/3734190

Three primary artifacts currently dominate the web design landscape: hand-drawn sketches, wireframes, and mock-ups [22, 30, 36]. A sketch serves as a preliminary, often rudimentary, representation of the intended UI design, enabling designers to swiftly consolidate and visualize their ideas. A wireframe, acting as a visual prototype, pinpoints the positioning of UI elements and content on the site. Lacking styling, graphics, or colors, it resembles a website's blueprint. A mock-up stands as a more refined and visually enriched version of a wireframe. It encompasses styling, graphics, colors, typography, and other intricate visual specifics. After obtaining approval for a wireframe or mock-up design from end users or other stakeholders, a web developer proceeds with the actual website creation.

Given that design and implementation typically fall under separate teams, the journey from concept to completion is not just time-consuming but can also be expensive. Such professionals invest immense effort in iterating, designing, and developing a site to meet end users' expectations. From an engineering interactive systems perspective, these challenges are more than technical hurdles; they underscore the importance of robust development pipelines that can streamline UI design, validation, and iteration with minimal back-and-forth between diverse stakeholders.

Considering these challenges, the concept of *automatic website generation* has arisen [8, 38]. It refers to the use of software and technologies to automatically produce websites without the need for manual coding. It integrates design and implementation phases, minimizing the need for back-and-forth adjustments between separate teams. By doing so, automatic website generation ensures a more direct generation of code from design intent. Consequently, the reduced iterations lead to quicker delivery times, cost savings, and fewer chances for miscommunication or errors that arise from repeated handoffs between design and development teams. Furthermore, the integration of automation allows for real-time adjustments based on immediate feedback loops. In traditional development scenarios, alterations post-deployment often require a revisit of the entire design-development-deployment cycle. With automatic generation, modifications can be made on the fly. Moreover, automatic website generation can help to democratize web creation. Those without a technical background or coding skills can still venture into designing and deploying professional-level websites. This may bridge the gap between developers and end users, while fostering a more inclusive digital landscape where creativity is not bounded by technical constraints. We posit that the problem of automatically translating intricate design artifacts into functional code requires not only algorithmic innovations but also an engineering methodology that integrates user-centric design, rigorous data curation, and real-world deployment concerns. This paper addresses that challenge by proposing a system capable of ingesting real-world design inputs and efficiently producing runnable prototypes, thereby advancing the engineering of fully interactive web applications. Within the predominant modalities for automatic website generation, the mock-up-driven approach, as the name implies, derives its functionality from mock-up designs and wireframes. Directly converting detailed mock-up designs or wireframes into functional UI code obviates the conventional, and manual transition from design to coding. The fidelity of the resulting websites can be considerably high, given the detailed nature of mock-ups, thereby ensuring a closer alignment with the designer's original intent. Similar to the mock-up-driven approach, which utilizes the most refined artifacts, the sketch-driven conversion method uses the most preliminary stage of design representation: hand-drawn sketches. The approach is particularly beneficial for novices; it offers an opportunity for those unfamiliar with web development processes to transform their basic sketches into functional websites. Sketches serve as a natural form of human-AI interaction because they harness the inherent human ability to visualize and express ideas through simple drawings, irrespective of technical expertise. This universal method of representation ensures intuitiveness and ease of use for a wide range of users, bridging the gap between imagination and digital creation. Moreover,

for designers, sketch-driven conversion allows them to rapidly test their interactive prototypes, making it easier to iterate and refine their ideas at a faster pace, as well as gain feedback.

Two modalities of mockup-driven and sketch-driven automatic website generation are possible: the *heuristic-based approach* and the *end-to-end approach*. The heuristic-based approach (e.g., [38] processes sketches and mock-ups by leveraging a set of predefined rules and patterns. In this approach, algorithms make determinations based on known patterns and guidelines. For instance, in the context of automatic web UI creation, a rectangular shape in a mock-up might be recognized as a button, and a series of parallel lines might be interpreted as text fields. The algorithm then generates the necessary code based on these identified patterns. Yet, this method has its limitations. Heuristic-based systems can be inflexible, and their effectiveness heavily relies on the quality and capabilities of the rules they are based on. If a design includes a novel element or a unique layout, the heuristic model might misinterpret it or fail to recognize it altogether. Moreover, it is a challenge to constantly update and maintain the ruleset as design trends evolve and as the complexity of designs increases. This means that while heuristic-based methods are generally efficient for more standardized and common designs, they may fall when faced with more intricate or innovative mock-ups. On the other hand, the end-to-end approach (e.g., [65]) adopts deep learning models to handle the entire process of converting a mock-up to a functional web interface. Instead of operating on a set of fixed rules, these models are trained on vast amounts of data, comprising various mock-ups and their corresponding web UI outputs. The larger, more diverse, and higher quality the training data, the better they become at making accurate predictions. The advantages of the end-to-end method are several. First and foremost, it can handle a broader array of designs, including those that might fail for heuristic systems. Given adequate training data, it can continually adapt and improve [31], keeping pace with evolving design trends. Furthermore, it can discern and learn subtle patterns in designs that might not be explicitly defined in heuristic rules. To illustrate, consider an unconventional mock-up where buttons are represented by ellipses instead of the typical rectangles. While a heuristic system might struggle to identify these as buttons due to its rule-based nature, an adequately trained end-to-end model could recognize them based on its exposure to diverse design patterns. Building on the innovations offered by deep learning architectures, end-to-end approaches have emerged as a powerful tool for turning mock-ups and wireframes directly into functional GUI code. The encoder-decoder framework, a common structure in these methodologies, traditionally leverages convolutional neural networks (CNNs) to parse image features, converting visual representations into intermediate language constructs, which are subsequently decoded to yield the desired code. Pix2code [7] was the first contribution introducing an end-to-end approach for the task; it is capable of translating web user interface screenshots and transcribing them into domain-specific language (DSL) representations, which can then be compiled into specific HTML code. While these deep learning models offer multiple advantages, their performance is primarily bottlenecked by the limited availability of specialized training datasets. Many of the currently available UI/code datasets lack diversity, often being overly simplistic and not adequately representative of the complexities encountered in real-world scenarios. This simplicity limits the true applicability of these models in practical settings. The models, having been trained on such datasets, do not generalize well when exposed to more complex, real-world designs, thereby hampering their effectiveness. Moreover, existing approaches to UI code conversion predominantly employ Long Short-Term Memory (LSTM)-based structures for decoding the visual representations and the potential of transformer-based architectures remains largely unexplored in this context. The attention mechanism, central to transformer models, can be especially advantageous for multimodal tasks as it learns to capture relationships between visual components and their corresponding code representations, potentially leading to more accurate design-to-code conversion.

The contributions of this paper are threefold and can be summarized as follows:

- **Web Scraping Pipeline:** To evaluate our system for automatic code generation from real-world web design, we introduce a specialized scraping pipeline to curate and clean online scraped code. By eliminating non-essential tags and scripts for visual appearance, our approach ensures that the resultant codes are less noisy. This, in turn, optimizes the transition from design mock-ups. Using the scraping method, we build our named "WebUI2Code" dataset consisting of 8,873 screenshot-code pairs.
- **Dataset Enhancement:** To overcome the limitations of existing datasets of mockups-code pairs, we generate a synthetic dataset of Bootstrap[1] websites that mirrors the diversity of web designs. Furthermore, we leverage it to adapt our model to enable direct code generation from sketches.
- **Architectural Evaluation:** Recognizing the limitations of traditional RNN-based methods for design-to-code tasks, we benchmark both a smaller multimodal transformer-based model (*Pix2Struct* [33]) and a larger-scale vision-language model (*FerretUI-Gemma2B* [63]) on our real-world and synthetic datasets. Results demonstrate that transformer architectures significantly outperform LSTM-based baselines [8], offering improved handling of complex UI layouts. Additionally, we highlight that while the larger Gemma2B model achieves slightly higher accuracy, it comes at a higher computational cost, thus underscoring the trade-off between resource efficiency and overall performance for large-scale web UI generation.

Through our work, we aspire to advance research in the realm of design-to-code transitions, with a special focus on real-world settings and designer needs. We believe that the methodologies we introduce can foster in the future a more accessible and efficient creation of web interfaces. The training and validation scripts and the datasets are available at https://bit.ly/3Rzpc0H.

2 Background and Related Works

Automatic website generation has emerged as a solution to ease the challenges associated with web design and development [7, 8, 38, 48, 54]. Automating the process makes it possible to quickly and efficiently create websites from design artifacts without wasting time on manual coding There are three primary methodologies underpinning automatic website generation: a) example-based, b) Artificial Intelligence-driven, and c) mock-up-driven.

Example-based automatic website generation allows users to create websites by referencing and adapting features from existing, professionally designed sites; by referencing real-world designs, it ensures quick customization and aesthetic appeal without the need for in-depth technical knowledge. Artificial Intelligence-driven website generation employs AI algorithms to design and build personalized websites based on user preferences, requiring minimal user intervention [2, 3, 12, 16, 24, 39]. Mock-up-driven automatic website generation transforms visual mock-ups or sketches of websites into working digital prototypes, often using heuristic techniques [5, 37, 45, 47, 50, 64] or deep learning models [21, 34, 65]; it ensures that the final product closely aligns with the initial vision.

Each method has its nuances. Mock-up-driven generation streamlines the transformation of a visual idea into a working prototype, but it requires a clear mock-up or further programming and designing effort in the case the code generation starts from wireframes or sketches [17]. The example-based approach provides more freedom to those less technically inclined, allowing for customization based on real-world examples, but it might not be as tailored or unique. On the other hand, while AI-driven tools are powerful and user-friendly, they might not always offer the depth of customization that professionals need, and the results might vary based on the sophistication of the AI algorithms [4, 11, 14, 20, 32, 62].

[1] https://getbootstrap.com, last visited on March 1, 2024

2.1 Example-based Automatic Website Generation

Many novice designers and developers, often with minimal web expertise, increasingly turn to pre-designed website templates. These templates help them establish aesthetically pleasing sites without delving into the complexities of code [19]. Though such templates offer customization options, including theme colors, font adjustments, and image uploads, their scope remains limited. Thus, some users might find them insufficient for their specific needs. Addressing this issue, researchers devised a method allowing those with little technical know-how to easily construct custom websites inspired by real-world, professionally designed sites. This system empowers beginners to explore these design exemplars, discern layout structures, select appealing elements and themes, and amalgamate these into their own designs through automated code generation.

Myers et al. [10] presented WebCrystal, a game-changing tool facilitating the extraction and replication of desirable HTML and CSS attributes from existing websites. The tool guides users with pre-set queries about adjusting HTML/CSS features. These questions come with textual descriptions, helping users choose the right attributes. WebCrystal then generates the suitable HTML/CSS code, which can be integrated directly or tweaked further. However, it has two main limitations: it is compatible only with HTML and CSS, neglecting client-side scripts like JavaScript, and occasionally, the extracted code might not perform identically across different sites. Hashimoto et al. [23] introduced a system allowing UI designers to probe for site designs mirroring their sketched layouts. Using a crawler, it amasses web pages into a database. When users sketch a desired layout, the system offers sites with similar designs. This aids novices in selecting and understanding HTML/CSS design structures. Still, due to database constraints and matching algorithm limitations, it does not always locate the perfect design match. Swire [29], geared towards mobile UI/UX design, lets designers explore Android UI designs resonating with their sketches or screen captures. Unlike the method by Hashimoto et al. [23], which relied on heuristic analysis, Swire uses a deep learning approach. However, Swire struggles with unique UI widgets or varied colors. Xiaofei [18] developed a tool that identifies Android apps resembling hand-drawn GUI sketches. It translates sketches into an intermediary language, applies deep learning to generate GUI frameworks, and then finds analogous apps from a database. While promising, it does not consistently provide perfect matches due to search strategy limitations. Behrang et al. [6] introduced GUIFetch, offering Java source code for Android apps closely matching users' GUI sketches. It involves two main stages: analyzing to detect potential Android apps and computing similarity scores between the sketched GUI and app GUIs. This tool aids in the tedious UI development phase, but it cannot detect or compare images.

2.2 AI-Based Website Builders

AI-driven website builders collect user preferences through a series of predefined questions [35]. Following this, they autonomously design and construct personalized websites based on the user's preferences, theme, and content, obviating the need for manual coding [35]. Consequently, even those without a background in web design or tech can establish their online presence using these platforms. A few pioneering commercial platforms are highlighted below: The Grid [41] was an AI web design tool that guided users through selections of color schemes, web components, fonts, and layout patterns. After inputting content, The Grid crafted the website, and users could recalibrate their initial style choices for further refinements. One drawback was its minimal design editing features [41]. Bookmark [9] is a cloud-based AI website platform featuring an AI Design Assistant (AiDA) that allows users to simply and quickly build responsive websites [42, 51]. Through a question-intensive first phase, it aims to meet the layouts and styles specified by the users. On top of this, it offers an expansive library of e-commerce and industry-specific templates [59]. Firedrop[43] specialized in building websites for small enterprises. By chatting with a virtual

assistant named Sacha, users could communicate their design preferences, which Firedrop then translated into the final design [35, 43]. Wix ADI (Artificial Design Intelligence) [57, 58], now evolved into the Wix AI website builder, has established itself as a leading contemporary AI website platform, offering affordability and customizability. It excels in producing websites by integrating optimal design layouts and components, often using a conversational interface [15, 58]. In 2018, Leia[26, 44] emerged as an AI website platform enabling the building of mobile-responsive sites initiated by simple voice commands or keywords [44, 53]. Zyro [28, 60], introduced in 2019 and now integrated into the Hostinger Website Builder [56], specialized in crafting SEO-optimized, responsive websites for small businesses with an array of template options and a user-friendly drag-and-drop interface [40]. Additionally, it offered complimentary AI tools like a logo maker, although its initial template range was somewhat restricted [60]. While these AI website platforms signify an important shift in web development, a comprehensive exploration of their underlying mechanisms and research is often not publicly accessible.

2.3 Mockup-driven Automatic Website Generation

A prevalent strategy to transform mock-up designs of websites into working prototypes harnesses *heuristic techniques*. The heuristic-based approach to website generation focuses on the use of domain-specific rules to guide the process. Such techniques typically extract web elements, discern their semantic relations, choose the fitting tags for these elements, structure the web elements hierarchically, and subsequently produce the source code. The method proposed by Huang et al. [48] identifies vertical and horizontal separators on the mock-up based on color differentiation. The resultant sections contain distinct web elements for which tags are generated. The tag generation uses the Random Forest method [27] for basic elements and a bottom-up approach for more complex elements. The heuristically proofed tags then inform the final website's HTML and CSS structure.

Shifting our focus to mobile app development, Nguyen et al. [2] introduced REMAUI, a method that automates UI source code from mock-ups for mobile apps. REMAUI's six-step process begins with the Tesseract OCR engine, which extracts text from screen captures. To rectify OCR's occasional misclassifications, domain-specific heuristics are employed. Alongside, computer vision methods identify UI boundaries to establish a UI hierarchy. REMAUI then refines this hierarchy, constructing a UI suitable for mobile apps. Upon testing, REMAUI averaged a 9-second runtime but faced challenges with certain OCR limitations and prototyping multi-page applications. P2A [39], in contrast, addresses REMAUI's limitation by prototyping animated mobile UIs from screen captures. Like REMAUI, P2A employs computer vision and OCR to detect UI widgets, but with the added capability of enabling users to add custom animations and transitions. After integrating these enhancements, P2A produces an executable with necessary asset files.

However, while heuristic methods can be accurate and efficient, they come with the limitation of the inherent imperfections of heuristic rules, which might not account for every scenario or outlier. Transitioning from heuristic, a distinctly different approach gaining traction in the domain of website generation is the use of *end-to-end methods*. These methods utilize deep learning models to transform website mock-ups or sketches directly into operational GUI code. They harness deep learning classifiers for converting visual layouts to code via an encoder-decoder setup. Notably, this strategy does not use preliminary image processing or heuristics but relies purely on the inherent capabilities of neural networks to interpret and translate visual designs. It is within this area that our contribution stands out, introducing innovative solutions to the existing challenges.

Pioneering this field, Beltramelli [7] introduced the "pix2code" model, leveraging a dataset of UI snapshots from iOS, Android, and websites; the method works by producing intermediate Domain Specific Language (DSL) code. This model includes one CNN and two LSTM networks. First, a CNN-driven vision model encodes UI captures into a fixed-length vector, an LSTM parallelly encodes the

DSL context into an intermediate representation. These vectors combine and are decoded using an LSTM, eventually classifying DSL tokens through a SoftMax layer. Several modifications of the architecture of Pix2Code [7] then emerged: Zhu et al. [65] introduced a model emphasizing UI components' hierarchical layout in the code. A CNN-based vision model extracts visual details from UI components, feeding a hierarchical LSTM decoder. With attention mechanisms, this model demonstrated a more accurate GUI code generation. Liu et al. [34] substituted LSTM with Bidirectional LSTM (BLSTM), improving the accuracy results on pix2code's dataset. Han et al. [21] aimed to create webpages with Cascading Style Sheets (CSS) styling details; their model utilized object detection methods and attention mechanisms to determine CSS contents.

Kumar [32] developed SketchCode, converting hand-drawn wireframes to HTML; leveraging the pix2code framework, it incorporated Gated Recurrent Units (GRUs) for encoding and decoding. Yong Xu et al. [62] crafted image2emmet, detecting GUI elements in web images, converting them to HTML-CSS; the tool integrated a Faster RCNN [46] and an LSTM, focusing on individual GUI elements instead of entire websites. Chen et al. [11] transitioned web mock-ups to mobile design code using a generative tool with RNN encoder and decoder and tested on 1208 real-world Android screen captures.

Building on the work of prior models, our research identifies and addresses the limitations often seen in RNN-based architectures, especially when considering their training dynamics and scalability. While RNNs have been foundational in the earlier stages of automatic website generation, the power of attention mechanisms, central to transformers architecture, remained untested in this domain. We thus leverage a transformer architecture capable of translating visual representation (images) into code tokens using an autoregressive encoder-decoder architecture. Our experiments demonstrate its superior performance over existing methods in the pix2code [7] Dataset. Beyond performance enhancements, this approach sets the stage for enhanced scalability, offering the promise of more sophisticated capable in future of automatic website generation research.

The power of the end-to-end approach comes from the ability of these models to learn from vast amounts of data and generalize to new, unseen data. End-to-end methods can sometimes produce more fluid and adaptive results due to their learning nature. However, they might require significant amounts of labeled data for training. Many current end-to-end solutions for Automatic Website Generation are benchmarked against Beltramelli's Pix2Code dataset. Such dataset, while beneficial for initiating research, might not sufficiently capture the complexities of real-world web designs. As a consequence, models trained solely on these datasets could be confined in their abilities and may not generalize well when faced with more intricate and diverse designs outside their training scope. Moreover, more complex proposed datasets, such as the one proposed by Chen [11], or RICO [13], focus primarily on mobile UI hierarchies, not on compilable HTML code. This highlights a significant gap in the available resources: a complex dataset of websites' code and mock-ups remains absent. The recently proposed WebUI dataset [61] does take a step in this direction by comprising scraped HTML code of the webpage. However, it brings along unnecessary tags and scripts that do not contribute directly to the reconstruction of the mockup. This excess of data can introduce noise and complexities, making the task of code generation from mock-ups more challenging than it needs to be. In response to this identified challenge, we introduce an HTML bootstrap synthetic dataset that, unlike the existing datasets, offers a wide array of components and layouts, targeting a diverse range of design elements to mirror different design scenarios, a detailed comparison of the distribution of components is shown in 6. We leverage this dataset to extensively train the transformer model. Given that the synthetic nature of our dataset allows precise component localization and labeling, we leverage the "synz" dataset [49]—comprising sketched web components by designers—to transform our dataset into a sketch-html dataset that we use to train a sketch-to-code system. Furthermore, in order to test our system on real websites

we introduce a web scraping pipeline specifically designed to clean the scraped code. By removing unnecessary tags and scripts, our processed data becomes more streamlined and better suited for code generation from mock-ups, overcoming the limitations of the WebUI dataset [61] for code generation from mockup. This approach ensures that the models are not misled by irrelevant code. We then utilize this method to collect a dataset 8,873 samples that we name "WebUI2Code" dataset, and we use it to test the capability of our model to generalize beyond synthetic sketches.

3 Method

We contribute three interconnected components in order to advance the automation of web development from visual designs and address the limitations of existing datasets and approaches. First, we implement a web scraping pipeline to collect and process real-world website code and screenshots. This pipeline retrieves the HTML rendered by websites and undergoes a sanitization process to remove undesirable code elements like scripts and comments. It also locates and downloads any associated CSS files, passing them through a custom parser.The processed HTML and consolidated CSS files are then rendered to capture website screenshots. A classifier filters these screenshots to retain only those preserving the original layout after code reduction.

Second, we implement a synthetic dataset generation method. This method procedurally creates websites adhering to a popular front-end framework, allowing control over parameters like layout, components, color palettes, and text. We integrate a recursive screenshot capture system to ensure complete rendering of all page elements. To create a sketch variant, we substitute UI components with hand-drawn sketches based on bounding box annotations.

Finally, we benchmark two transformer architectures for the design-to-code task: a Pix2Struct model and a larger, more powerful Gemma2b model. Both models integrate a vision transformer encoder—which processes rendered website screenshots—with an autoregressive transformer decoder that sequentially generates the corresponding HTML and CSS tokens. We evaluate these architectures on a real-world dataset obtained from our web scraping pipeline and on synthetic datasets under multiple conditions. These conditions include scenarios where rendered elements are replaced with sketches to simulate lower-fidelity designs, variations in website layout complexity, and adjustments in token thresholds for code generation. This evaluation allows us to assess the performance differences between the smaller Pix2Struct and the larger Gemma2b model across diverse design-to-code scenarios.

3.1 WebUI2Code Dataset: Real-World Website Screenshot-Code

Our data collection procedure draws upon a multi-stage pipeline to retrieve, sanitize, and validate websites, ensuring both high structural fidelity and practical manageability for subsequent machine learning tasks. As illustrated in Figure 1, the process begins by fetching each target URL in headless mode via Selenium and Google Chrome. The browser runs at a fixed resolution (1280×1024) to standardize captures. A retry mechanism is triggered whenever a site fails to load fully. Common pop-up dialogs (for example, cookie consent banners) are automatically dismissed, either through the removal of their scripts or by simulating clicks on typical acceptance buttons.

Once the raw HTML is obtained, an automated sanitizing phase follows, where extraneous or dynamic tags—such as `script`, `meta`, `noscript`, `svg`, and `iframe`—are systematically removed. The sanitization script further rectifies malformed tags and substitutes asynchronous attributes (e.g., `data-src`, `data-lazy-src`) with their standard equivalents (`src`, `srcset`). This step also replaces all external images with a uniform placeholder image to avoid the overhead of resource downloads and to maintain consistent visual references. To reduce trivial layout variability, certain HTML elements (`<ol>` tags, for instance) are converted to functionally similar ones (such as `<ul>`) when no essential structural difference is detected. Following this filtering, a cleansing tool (`clean-html`)

removes any remaining comments, erratic whitespace, or superfluous line breaks, then formats the code with a consistent indentation scheme.

With the cleaned HTML in hand, the pipeline locates external CSS references, downloading each `.css` file and parsing it via a custom extension built atop `tinycss2`. This parser identifies at-rules and qualified rules, allowing selective removal of styles that do not match any tags or classes found in the sanitized HTML. Properties considered strictly decorative or irrelevant to the website's core structure (such as transition animations or legacy browser-specific rules) are pruned to minimize complexity. The remaining valid CSS is then merged into one consolidated file, and references in the HTML are updated accordingly.

Before final screenshot capture, we apply a *web framework detector* that checks for traces of frameworks like React, Gatsby, Nuxt, Backbone, or Next. Empirically, these frameworks introduce significant client-side code that often fails to render properly after sanitization. Whenever such frameworks are detected, the pipeline excludes the corresponding site from our final dataset to preserve consistency. As an additional filter, any site whose sanitized HTML has zero CSS classes or is trivially empty is likewise removed.

Following these exclusions, we generate screenshots by loading the local HTML file in the same headless browser environment. By rendering the sanitized HTML and merged CSS, the captured screenshot more accurately reflects the final "minimal" version of the webpage than if we had captured the original online page. Figure 2 shows examples of the resulting mockups. The average degree of reduction is substantial, particularly in the CSS: Table 2 shows that many pages go from tens of thousands of lines to a few thousand or even fewer.

A final quality check employs a ResNet50 convolutional neural network [25] fine-tuned on a labeled set of "good" vs. "bad" screenshots. This binary classifier learned to distinguish reliably between pages that preserve essential layout and those that exhibit severe structural breakage or missing styles. Its training set derived from earlier pilot experiments, in which we manually assigned grades (0 to 5) based on visual fidelity and structural completeness. Poor scores typically correlated with blank pages, severely distorted layouts, or frameworks that the pipeline could not handle. By converting these manual labels into "good" and "bad," the classifier reached over 80% accuracy on a held-out test. After this final classification step, "bad" images are discarded, leaving a curated collection of "good" screenshots and their processed HTML/CSS for subsequent research.

Three preliminary studies tested this pipeline on subsets of varying complexity: an initial experiment on a small list of 51 blog sites, a second on 100 sites from the Majestic Million ranking, and a third on 100 `.blog` domains. These incremental trials highlighted not only the pipeline's strengths (notably for simpler or blog-oriented pages) but also the presence of difficult cases (e.g., large commercial sites with anti-scraping measures or elaborate client-side rendering).

	Total
Errors	29736
Excluded	16459
Bad images	19716
Good images	34089
TOT	100000

Table 1. Results of filtering websites based on quality.

Human evaluators compared the processed screenshots to their original online counterparts, assigning numeric grades and analyzing major differences. The results informed thresholds for excluding particular frameworks or "unstructured" pages. When extended to 100 000 domains from

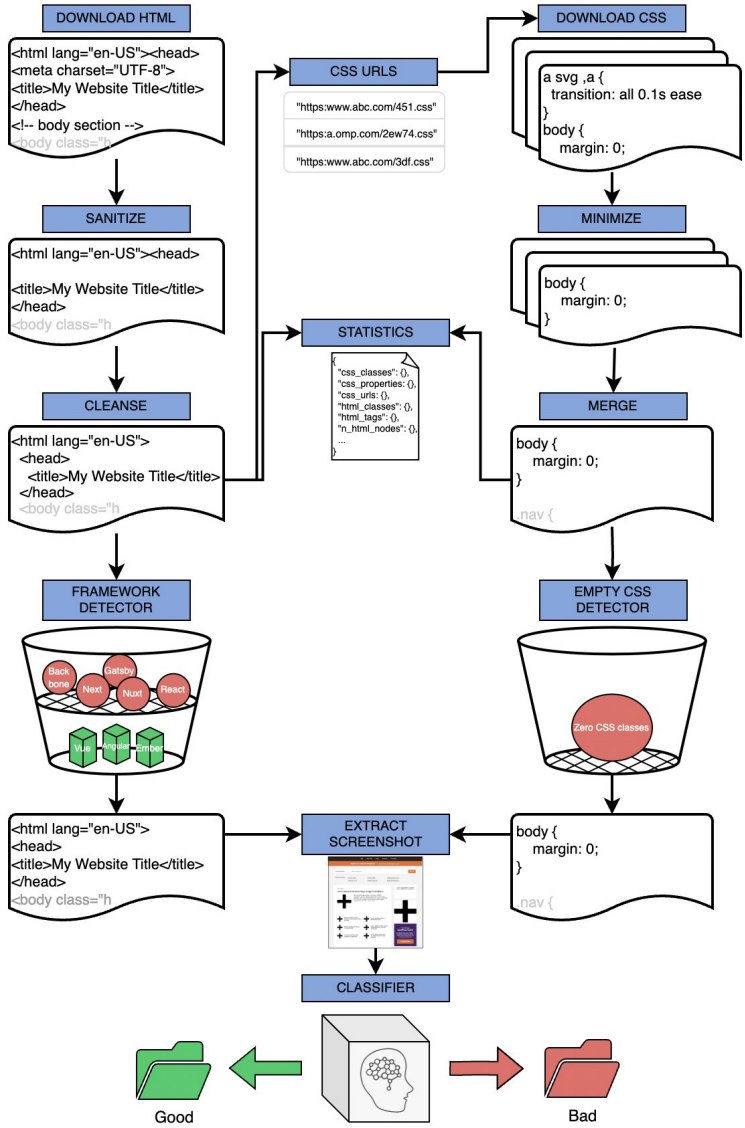


Fig. 1. The diagram illustrates the step-by-step process of obtaining website code and screenshots. Initially, the HTML file is downloaded, sanitized, and cleansed. Next, the HTML file is processed by a web framework detector that filters out files containing specific frameworks. Following this, CSS URLs are extracted from the HTML file, and the corresponding CSS files are downloaded, minimized, and merged. A detector is then used to exclude files that have zero CSS classes. Subsequently, the processed HTML and CSS files are used to extract website screenshots. Finally, the screenshots are labeled based on their quality by a classifier.

the Majestic Million, the pipeline retained approximately 34 000 "good" pages out of 100 000 total entries; see Table 1 for an overview. Overall, the biggest sources of error were connection and SSL issues, especially from highly-protected websites, along with malformed CSS.

By systematically refining the pipeline at each stage—from HTML retrieval and sanitization, through CSS minimization and screenshot classification—our WebUI2Code dataset arrives at a

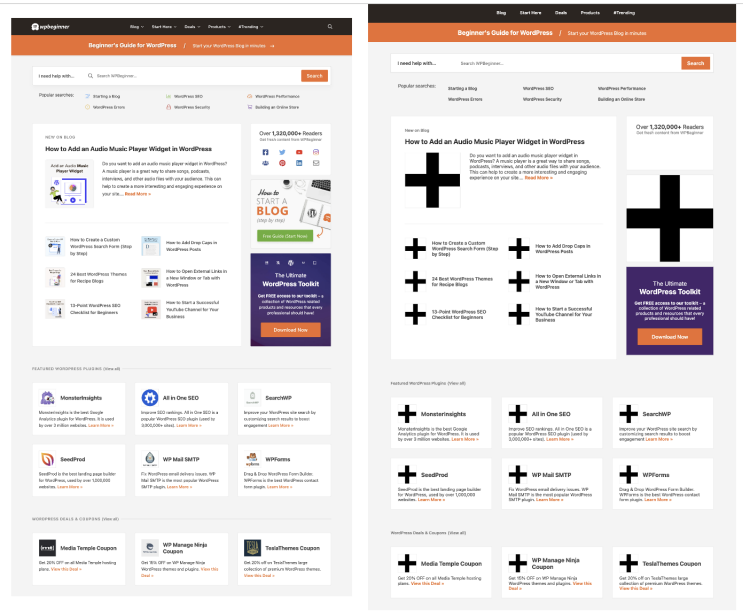


Fig. 2. Screenshot of website before and after processing.

relatively clean, structurally representative set of real-world UIs. This multi-tiered filtering strategy intentionally reduces coverage to guard against visually unrecognizable or fundamentally broken pages. The final result is a large-scale corpus of HTML/CSS/screenshot triples that are significantly less noisy than raw scrapes, offering an accessible testbed for design-to-code models and other interface-centric research.

For a detailed dive into our methodology for acquiring and processing website codes, we direct readers to Appendix A. Alongside these procedural insights, the appendix showcases various statistics pertaining to the analyzed websites, offering a perspective on the makeup of our dataset for readers to gain a clear understanding of our methodological choices and their implementation.

	Raw	Processed
CSS Classes	1788.03	143.39
CSS Classes Skipped	0	1596.25
CSS Properties	220.31	78.42
CSS Properties Skipped	0	17.83
CSS URLs	8.14	8.11
HTML Classes	238.64	234.75
HTML Tags	35.94	26.86
HTML Nodes	860.75	699.58
Lines CSS	20240.5	2255
Lines HTML	1542.25	996.08

Table 2. Summary of collected statistics on average for each website

3.2 HTML Bootstrap Synthetic Dataset

The HTML Bootstrap Synthetic Dataset was built using the open-source tool WebGenerator [52]. This tool was developed to create synthetic web-based UIs using the Bootstrap framework. The tool comes with a variety of generation options, such as probabilities of layouts and sections, screenshot sizes, and components. These components include commonly used items like Cards, Placeholders, Tables, and Navigation bars, as well as specialized components like Carousels and Forms. To populate these elements, the tool utilizes random "lorem ipsum" sentences, simulating how such UIs might appear in practical applications. Another feature of the tool is its ability to generate a variety of website styles by selecting random color palettes and modifying the associated CSS file in the HTML code, allowing for multiple distinct website designs. By manipulating the parameters of the WebGenerator, the generated websites have a broad range of designs and structures. This diversity ensures that any model trained on this dataset will be more robust and generalizable. Moreover, since the dataset uses the Bootstrap framework, it holds relevance in the modern web development scene. Bootstrap is among the most popular front-end libraries, and its components are often found on many websites across the Internet. Therefore, it has practical significance and can be used for research that aims to produce real-world applications. To adapt the tool to our needs, we introduced a recursive system for screenshot capturing that ensures that all the webpage elements are contained in the screenshot.

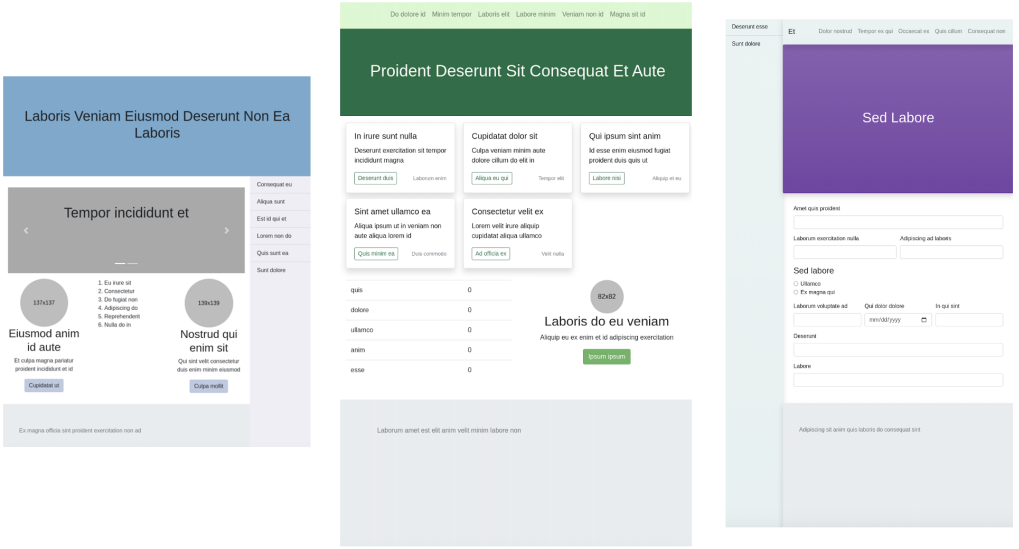


Fig. 3. A series of random sampled web interface mock-ups from synthetic dataset demonstrating varied design patterns. From left to right: a content-focused layout with distinct navigation elements; a grid-based interface highlighting data presentation; and a minimalistic approach with prominent call-to-action features.

Samples of the website screenshots produced by the WebGenerator are illustrated in Figure 3. Since these are synthetically generated, they do not contain any real-world information, making them an ideal dataset for various research applications without concerns about privacy or data usage rights. The accompanying JSON annotations detailing regions of the GUIs and their corresponding type provide researchers with a detailed understanding of the structure and design elements used

in the webpage. Such annotations can aid in supervised machine-learning tasks where precise labels are required. We introduced modifications in the component annotation procedure. Instead of providing general labels like 'header' or 'footer', we identified more fine-grained components such as text and buttons. This modification facilitated the subsequent conversion to sketches. Once configured, we generated a dataset comprising 50,000 samples. Each of these samples contains a PNG image, which is a screenshot of the website, its associated HTML code, and JSON annotations.

3.3 Sketches Synthetic HTML Bootstrap Dataset

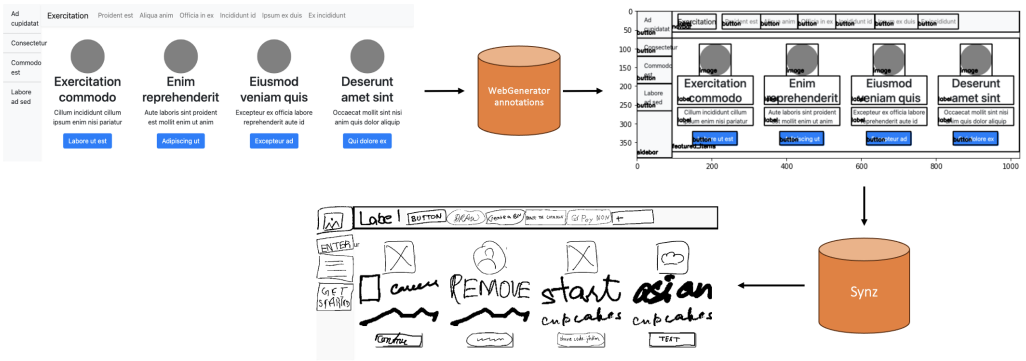


Fig. 4. This flow diagram provides a representation of the multi-stage transformation process starting with synthetic webpages that contain structured design elements and textual content. Specific components are extracted and translated into generator annotations. The annotations then serve as a bridge to transform the structured designs into more abstract, hand-drawn sketches.

To utilize the synthetic dataset for the sketch-to-code challenge, we augmented the generation function with a method to monitor bounding boxes of fine-grained components, the foundational design elements. Thanks to the dataset's synthetic nature, we could easily annotate and retrieve these bounding boxes without relying on object detection methods. Following this, the identified components were substituted with their sketched versions from the Synz dataset [49], a comprehensive collection of sketched web components. Each component is cropped to the smallest rectangle that includes the first non-white pixel since we observed that, in the Synz dataset, there is a redundant whitespace. Furthermore, out of the top 10 components with a similar aspect ratio to the bounding boxes, one is randomly selected for use. This approach enabled the creation of a synthetic dataset of web sketches with diverse compositions. As an outcome, this version produced nearly 10,000 websites (specifically 9,789), which adhered to the same methodology and parameters as the original synthetic Bootstrap dataset.

3.4 Transformer Architecture for Design-to-Code

Transformer models have reshaped sequence-to-sequence translation tasks by using self-attention to capture long-range dependencies without the recurrence bottleneck of LSTM-based approaches [55]. For design-to-code translation, a transformer can ingest image patches (visual tokens) as well as partial code tokens (textual context), generating new tokens that represent the evolving HTML or DSL output (Figure 5). Below, we briefly summarize our adopted approach, emphasizing practical details relevant to handling large screenshots and lengthy code sequences.

Vision Transformer Encoding. Following Pix2Struct [33], the input screenshot is first divided into equally sized patches (e.g., $P \times P$ pixels), each linearly embedded into a latent vector. Stacking transformer layers over these patch embeddings provides a contextualized representation of the webpage image, capturing both global layout and local component details. Unlike purely convolutional backbones, this patch-wise approach inherently treats the layout as a sequence of visual tokens, enabling flexible handling of different aspect ratios and UI complexities.

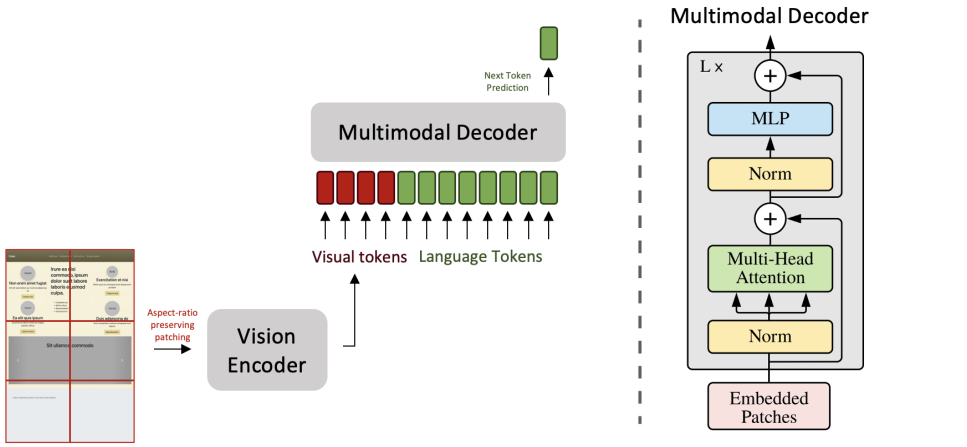


Fig. 5. A visualization of the process to convert web mock-ups into code. On the left, a web mock-up is segmented into patches, which are processed by Vision Transformers into contextualized encodings. The center highlights the autoregressive transformer decoder, generating code tokens sequentially based on visual encodings.

Sliding Window for Long Code Generation. Generating the full HTML of a page often exceeds common context-length limits for Pix2Struct [33]. We therefore apply a *sliding window* strategy on the text side: at each step, the decoder receives (i) the entire set of image embeddings, and (ii) a truncated history of previously generated tokens (e.g., the last N tokens). It then generates the next M tokens, shifts the window, and repeats. Empirically, we find this process remains coherent because the model implicitly learns to continue from where it left off, associating a specific image region with partial code segments. Although each window does not include the complete code history, local textual context plus the consistent image encoding suffice to preserve syntactic consistency. This mechanism effectively sidesteps GPU memory constraints while still enabling reconstruction of long HTML files. FerretUI-Gemma, in contrast, can accommodate larger textual contexts in a single forward pass, thus relying less on sliding windows and more on an extensive memory of the entire code sequence. While this often yields better performance on very large or heavily annotated web pages, it requires substantially more computational resources. By contrast, our Pix2Struct-based approach plus sliding-window text decoding strikes a balance, allowing complex layouts to be handled without exceeding memory limits. Both methods underscore the importance of accurately modeling *image-to-code ratio awareness*: the network must align individual parts of the UI screenshot with semantically corresponding code regions, whether that alignment is

done incrementally (sliding window) or in a single broad sweep (long context). As shown in our experiments, both designs can be effective, but practitioners should weigh the trade-off between hardware resources and the complexity of web UIs under consideration.

4 Experiments

In the subsequent sections, we evaluate the performance of the transformer-based Pix2Struct architecture across three distinct scenarios: the validation of our real-world WebUI2Code dataset and synthetic datasets through the model's performance, and the assessment of Pix2Struct's effectiveness for design-to-code generation compared to traditional RNN-based methods on existing benchmarks. These experiments provide insights into both the potential and limitations of transformer-based approaches for automated web development, with implications for practical web design applications.

4.1 Datasets

The experiments utilized the following datasets to evaluate the proposed method:

4.1.1 WebUI2Code Dataset: The "WebUI2Code" dataset, described in Section 3.1, emerges from our scraping pipeline designed to extract website codes and their corresponding screenshots from the web. Comprising 8,873 samples, each data point in this dataset consists of an HTML file, an associated CSS file, and a screenshot. Alongside these, the dataset also contains supplementary files, such as a JSON detailing the metrics from the scraping process, and the unprocessed versions of the HTML and CSS files. The WebUI2Code dataset was segmented into distinct versions to adapt to different experimental needs, grounded in the dataset's size and complexity. By creating subsets of the data with specific token thresholds, we aim to ensure that the experiments are both manageable and scalable.

The WebUI2Code-4096 Version has been set with a token threshold of 4,096. This version includes 2,442 samples and has been curated to align with the token constraints applied to other datasets in parallel studies. Next, the WebUI2Code-8192 Version increases the token threshold to 8,192, accommodating slightly more intricate samples. Composed of 1,906 samples, the number of usable samples in this version varies based on additional constraints and preprocessing measures that may be applied.

The dataset continues to expand with the WebUI2Code-12288 Version, which is characterized by a token threshold of 12,288. With 2,170 samples, this version showcases a broader range of web designs, representing the diversity of web content as the token limit grows.

Lastly, the WebUI2Code-16384 Version presents a more comprehensive collection of web designs, accommodating a token threshold of 16,384. Consisting of 2,355 samples, this version has been crafted to include websites with more extensive content and structure. Each of these versions represents a distinct facet of web design, starting from simpler web pages to more content-rich and complex designs. As the token threshold increases, the datasets not only grow in sample count but also in the richness and diversity of content, offering a graded approach for model evaluation.

4.1.2 HTML Bootstrap Synthetic Dataset(s): The HTML Bootstrap Synthetic Dataset, as described in Section 3.2, introduces a more complex ground for testing and evaluation of recent encoder-decoder architectures. This dataset includes a variety of design elements such as cards, placeholders, tables, navigation, carousels, forms, and others. Synthetically generated web pages are designed using a set of predefined templates that mimic real-world web design conventions. These templates are chosen to cover a wide spectrum of web page layouts. By incorporating such diversity, the Synthetic Dataset serves as a valuable resource for benchmarking the performance of algorithms intended to understand and generate web code from visual inputs. The result is a dataset composed of 50,000 samples, each one consisting of a screenshot of the website and its HTML-Bootstrap code. A variant

of this dataset was created having sketches of website components instead of the rendered elements. This was done by identifying the area that each real element occupies and substituting it with a sketched version of it, as described in Section 3.3 The result is a dataset of almost 10.000 websites (9,789), whose HTML codes are created in the same way and with the same parameters as the original Synthetic Bootstrap dataset.

4.1.3 Pix2code Dataset: The Pix2Code dataset [7], while relatively simple in its composition, serves as our baseline for comparison with existing literature. It comprises screenshots of various web and mobile interfaces, each paired with a Domain Specific Language (DSL) code. This DSL acts as an intermediary language detailing the UI's structure and components and facilitating its conversion into functional codes like HTML or Android XML. This dataset is divided into three categories: Android, iOS, and Web interfaces. The web section, which is the focus here, contains 1,742 samples. A distinctive feature of this dataset is its streamlined design. It incorporates only 12 unique structural elements, and the UI codes primarily consist of these elements and specific arrangement indicators. Elements such as "small-title", "text", and "quadruple" are prevalent, representing nearly half of all dataset elements. On average, a website's code in the dataset has between 8 to 56 elements. Conversion of the DSL codes into HTML led to a variant of the dataset, pairing these HTML codes with screenshots. A difference arises in text between the original screenshots and the compiled ones due to the non-deterministic nature of character generation during the compilation process. An iteration of this dataset was introduced, using 'lorem ipsum' as placeholder text, aiming to maintain consistency. Historically, the Pix2Code dataset has been utilized in various studies, aiding in the enhancement of certain methodologies. Nevertheless, its potential as a benchmark for future developments might be reaching its limits.

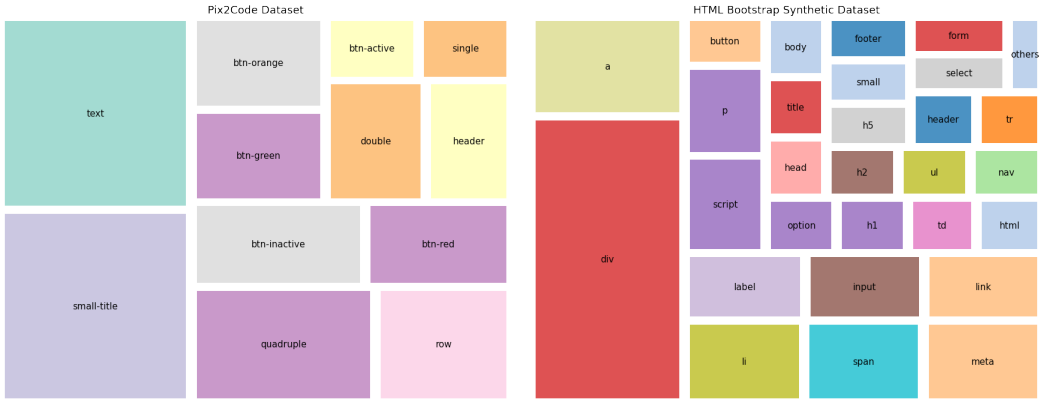


Fig. 6. Treeplot comparison between the distribution of tags between a) Pix2Code dataset and b) the HTML Bootstrap Synthetic Dataset. The area of the rectangles represent the distribution of the components in the dataset.

4.2 Metrics

Several metrics were utilized to assess a model's capability in predicting website code from screenshots. Textual metrics predominantly relied on the raw versions of responses and predictions, while some, like structural BLEU and HTML tree edit distance, required complete code compilation and post-processing. Errors identified during corrections were stored for later analysis, and codes rendered by browsers were used to derive screenshots for image similarity metrics.

BLEU. BLEU is a widely-accepted metric for evaluating machine-translated text. It gauges the overlap of n-grams between generated and reference texts. However, its applicability in code prediction is limited due to code's syntactic nature. Given this, a variant termed "structural BLEU" was introduced, emphasizing structural elements of code by removing non-structural elements. The Natural Language Toolkit (NLTK) was employed to implement the BLEU score, utilizing a specific smoothing function to address precision disparities in shorter texts.

Edit Distance. This metric measures the Levenshtein distance between two texts, denoting the character modifications needed to equate them. For instance, converting "rain" to "shine" necessitates three modifications. The NLTK was again utilized for this metric's implementation, with all operations assigned equal costs. A normalized variant, based on the maximum character count between response and prediction, was also used.

HTML Tree Edit Distance. Driven by the motivation behind "structural BLEU", a distance metric emphasizing structural elements was introduced. HTML code is represented as a tree structure, with the tree edit distance calculated using the Zhang-Shasha algorithm. The algorithm determines the minimum node modifications to transform one tree into another. The Beautiful Soup parser extracted HTML nodes, and a Python rendition of the Zhang-Shasha algorithm was utilized. A normalized version, factoring in the maximum node count between the two trees, was also considered.

Structural Similarity Index (SSIM). Unlike other methods, SSIM highlights pixel inter-dependencies, offering insights into a visual scene's structure. The scikit-image version of SSIM was utilized. A limitation is its requirement for identical image dimensions, necessitating resizing, which can inadvertently shift image components, affecting the SSIM index.

4.3 Configuration

All experiments were conducted on a dedicated server equipped with four NVIDIA A100 GPUs (each with 68 GB of VRAM). This setup provided the necessary computational capacity for large-scale transformer training on both our Pix2Struct and FerretUI-Gemma2B models. For initial testing and prototyping tasks, we also employed smaller-scale GPU resources (NVIDIA V100s with 32 GB VRAM or T4s with 16 GB VRAM), striking a balance between computational performance and cost efficiency.

Pix2Struct. We used a two-phase training schedule. In the first phase, we trained on the synthetic dataset for 20 epochs with a learning rate of 1×10^{-4}. In the second phase, the model was fine-tuned on the WebUI2Code dataset for 10 additional epochs using a learning rate of 1×10^{-5}. Our batch size was set to 32, with gradient accumulation over 4 mini-batches to effectively simulate a larger batch size of 128. We employed Adafactor as our optimizer and used a cosine learning rate schedule, incorporating a 5-epoch warm-up period. We also applied gradient clipping with a maximum norm of 1.0 to ensure stable training. For processing high-resolution screenshots, we restricted image inputs to a maximum of 1024 patches while preserving aspect ratio. As for text decoding, we used a *sliding window* strategy on the HTML tokens: each window spanned 1024 tokens, with a 256-token overlap to give context from previously decoded chunks. Empirically, this approach alleviated memory constraints and allowed the model to autoregressively piece together longer code sequences with minimal coherence loss.

FerretUI-Gemma2B.. Training Gemma2B followed a similar 4×A100 GPU setup, but with parameters specifically adapted to large-scale vision-language instruction tuning. We used a per-GPU batch size of 2 (for a total of 8 effective across 4 GPUs), gradient accumulation steps of 2, and ran for 10 epochs at a learning rate of 2×10^{-5}. To accommodate potential variability in input

image resolutions, Gemma2B was configured with an "anyres" setting, resizing images to 336×336 while maintaining aspect ratio metadata. The architecture and script parameters (e.g., gradient checkpointing, half-precision BF16, warm-up ratio of 0.03, and a cosine scheduler) were optimized for training stability over large instruction-tuned configurations. Although Gemma2B supports more extensive context in a single forward pass than Pix2Struct, it incurs a higher computational cost and VRAM footprint due to its larger parameter count.

With both models, we used a 90:10 split for training and testing, reserving a small portion of the training set (approximately 10% of the total samples) for validation and hyperparameter tuning. We performed evaluations every 5 epochs for Pix2Struct and at 100-step intervals for Gemma2B, logging metrics such as cross-entropy loss and BLEU scores. Together, these configurations enabled us to explore trade-offs between memory consumption, training speed, and performance on real-world datasets.

5 Results

We benchmarked two different transformer-based approaches—the comparatively smaller *Pix2Struct* model and the larger *FerretUI-Gemma2B*—alongside an LSTM-based method (Pix2Code) across multiple datasets to assess code-generation quality. Table 3 summarizes our findings, reporting BLEU scores, edit distance measures, and visual appearance metrics (SSIM).

Model/Dataset	BLEU (avg) ↑	ED (avg) ↓	N.ED (avg) ↓	SSIM (avg) ↑
LSTM				
Pix2Code	0.878	4.925	0.132	0.935
Pix2Struct				
Pix2Code	0.983	4.437	0.016	**0.942**
Synthetic Bootstrap	**0.929**	**443.890**	**0.081**	0.783
Synthetic-Sketches Bootstrap	**0.825**	**1146.678**	**0.202**	0.810
WebUI2Code-4096	**0.436**	6920.180	0.714	0.547
FerretUI-Gemma2b				
Pix2Code	**0.995**	**1.196**	**0.004**	0.939
Synthetic Bootstrap	0.740	1796.909	0.226	**0.885**
Synthetic-Sketches Bootstrap	0.725	1757.678	0.242	**0.879**
WebUI2Code-4096	0.397	**5790.563**	**0.487**	**0.978**

Table 3. Comparison of Metrics Across Different Datasets

We begin with the Pix2Code dataset to compare the baseline LSTM architecture against transformer approaches. As shown in Table 3, Pix2Struct attains a BLEU score of 0.983, comfortably surpassing Pix2Code's 0.878. The normalized edit distance (N.ED) also drops sharply from 0.132 to 0.016, indicating that the transformer's output is far closer to the reference code. Although the SSIM values are comparable (0.935 vs. 0.942), Pix2Struct shows a clear advantage in both lexical (BLEU) and structural (N.ED) comparisons, underscoring the benefits of attention-based models for design-to-code translation.

On the Synthetic Bootstrap and Synthetic-Sketches datasets, Pix2Struct achieves high BLEU scores (up to 0.929) while still maintaining reasonable SSIM values (e.g., 0.783). In the case of sketches—where text is replaced with drawn elements—BLEU values decrease slightly, reflecting the added complexity of parsing purely visual inputs. Nevertheless, the model continues to capture structural layout effectively (as indicated by relatively low edit distances and a strong SSIM of

0.810). These results confirm that attention-based methods can generalize to varied input styles, from straightforward text elements to purely visual sketches.

We also evaluated FerretUI-Gemma2B, a larger vision-language architecture, which offers the potential for stronger multimodal embeddings at the expense of higher computational demands. As shown in Table 3, Gemma2B achieves slightly lower BLEU scores on Synthetic Bootstrap (0.740) than Pix2Struct (0.929), but it notably yields higher SSIM on the same dataset (0.885 vs. 0.783), suggesting that while token-level agreement may dip, Gemma2B more faithfully reconstructs the visual layout. On real-world websites from *WebUI2Code-4096*, both models post BLEU scores around 0.436. However, Gemma2B delivers a much stronger SSIM of 0.985, whereas Pix2Struct remains at 0.547. This difference highlights Gemma2B's advantage in replicating the visual appearance of complex interfaces, although it requires substantially more computational resources.

Despite strong results on synthetic and sketch-based datasets, both transformer approaches face challenges when tackling the intricate, idiosyncratic HTML structures found in real-world websites. Even with sliding-window decoding to handle longer HTML sequences, the average BLEU scores hover around 0.43–0.44. Observed failures include misinterpreting complex CSS usage or generating inconsistent tag hierarchies when the code extends beyond typical patterns. Notably, FerretUI-Gemma2B often compensates for lexical mismatches with strikingly faithful visuals (leading to a high SSIM), illustrating that different models may trade off textual accuracy for layout fidelity.

Our experiments confirm that transformer-based methods substantially outperform the LSTM baseline on standard benchmarks (Pix2Code) and exhibit robust performance on synthetic datasets of diverse complexity. Pix2Struct generally yields higher BLEU scores, whereas FerretUI-Gemma2B shows better performances in replicating visual appearance (SSIM, ED) but comes with greater computational cost. Figure 7 shows illustrative samples from each dataset.

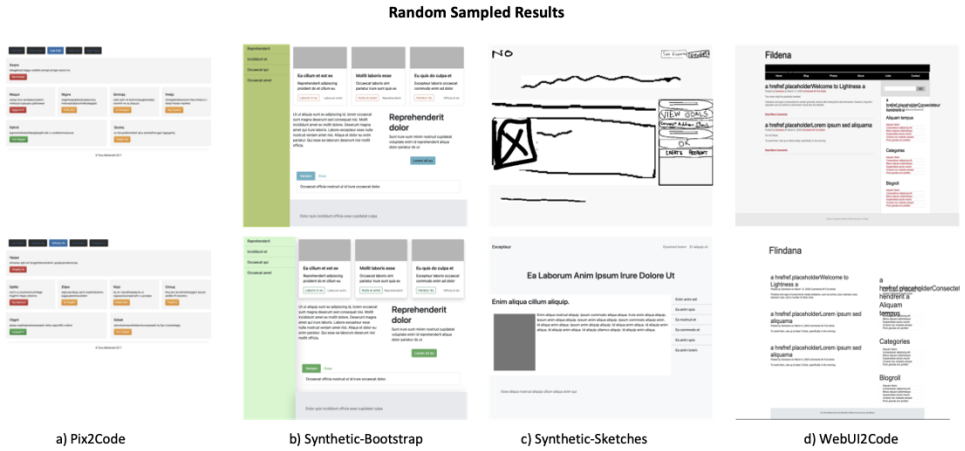


Fig. 7. Random samples from our data (first row) and predictions (second row).

6 Discussion

Our experimental results reveal three key findings about transformer-based approaches for web design automation. First, the transformer architectures significantly outperforms traditional RNN-based methods on established benchmarks, achieving superior accuracy in code generation from simple UI mockups. This validates the potential of these architectures for the design-to-code task, aligning with their success in other vision-language domains.

Second, the models demonstrate robust performance on our synthetic datasets, suggesting its viability for handling varied design representations and components. This capability, particularly in processing both high-fidelity mockups and sketches, indicates strong potential for supporting different stages of the design process and enabling practical design support tools.

The third finding emerges from our experiments with real-world websites in the WebUI2Code dataset, where we observe a significant performance drop. Several factors contribute to this challenge: the inherent complexity and variability of production websites compared to synthetic data; the presence of dynamic elements and complex CSS styling that may not be captured in static screenshots; and the diverse coding practices and optimization techniques used in real-world development that create many-to-many relationships between visual appearance and underlying code. Additionally, real websites often contain intricate responsive design patterns and browser-specific optimizations that are difficult to infer from single screenshots.

These findings have important implications for the future of automated web development. While transformer architectures show promise for design support tools and prototyping assistance, the gap between synthetic and real-world performance suggests that current models might struggle to rely on real-world datasets for training. However, models trained on synthetic data can still provide valuable support as augmentation tools for tasks like rapid prototyping, design exploration, and learning assistance where perfect code generation is not required. Furthermore, these advancements could democratize web development by enabling users with limited coding skills to transform their designs into functional code, while the sketch-to-code capabilities specifically enhance rapid prototyping and early-stage design exploration. However, the successful adoption of these technologies will depend not only on continued improvements in model architectures and training data but also on their thoughtful integration into existing design workflows and tools to maximize their benefit for end users.

6.1 Implications for Engineering of Interactive Systems

Transforming UI designs into functional prototypes through automated code generation holds significant promise for both novice and expert users of interactive systems. For end users, the ability to sketch an interface and obtain a functioning model considerably lowers the barriers to technology creation. Such an approach aligns with the broader goals of user-centered design, where empowerment and ease of iteration are key. Instead of requiring deep knowledge of implementation details, users can rapidly test, refine, and validate their ideas in tangible form. This heightened accessibility could motivate wider participation in UI creation, fostering communities of practice in which design ideas flow more freely and collaboratively.

Expert developers and designers, conversely, often find themselves constrained by deadlines and the necessity to evaluate multiple design variants. By introducing a design-to-code pipeline based on transformer architectures, these professionals can offload the more repetitive or mechanical aspects of building a prototype. The system's learned capability to generate coherent code scaffolds from high-level concepts accelerates the overall workflow. In traditional development cycles, even small changes to the interface design might demand a cascade of manual coding and layout adjustments; here, the modifications are handled at the conceptual level, with fine-tuning mechanisms absorbing

the burden of lower-level coding. Such gains in efficiency promise not only a faster route to polished prototypes but also a means of focusing human expertise on high-impact creative and user-centered tasks.

Additionally, these methods create potential pathways for integrating directly into contemporary design toolchains. In many teams, interactive design is first captured by dedicated software—ranging from wireframing platforms to more specialized prototyping tools—before moving into code. The vision of linking these platforms to a robust transformer model means that the end-to-end process from concept to interactive demonstrator can be significantly shortened. Iterative user testing, an essential component of human-computer interaction research, also benefits from this streamlined approach: because prototypes can be generated and adapted at speed, user feedback loops can be conducted more frequently, and the design refined in near-real time. Ultimately, this fusion of user-centric design principles with powerful language-modeling techniques stands to reshape how interactive systems are engineered, democratizing access for newcomers while enhancing productivity for seasoned professionals.

6.2 Adaptive Learning and Personalization

As AI systems increasingly guide web design and development, they are no longer confined to static, one-size-fits-all outputs. Instead, emerging approaches can adapt to individual user behavior patterns, offering personalized code-generation experiences. By analyzing a designer's past decisions, a modern AI platform can learn to anticipate future preferences and suggest UI components or design layouts that align with the user's evolving style. Such adaptability builds upon advanced algorithms: reinforcement learning allows a system to iteratively adjust its parameters based on real-time feedback, while transfer learning enables knowledge acquired in one domain—such as layout heuristics—to benefit design tasks in another. From a human-computer interaction perspective, these adaptive AI systems intersect with both End-User Development (EUD) and Adaptive User Interfaces (AUIs). Through EUD, non-technical individuals can "program" the AI to focus on certain design constraints or branding elements without formal coding expertise. Meanwhile, AUIs dynamically reconfigure their interactions based on user feedback, supporting diverse skill levels and interface preferences. Together, these elements promote inclusivity and foster user empowerment, ensuring that automated design tools remain responsive to each designer's creative direction. Our contribution forms a foundation for these adaptive workflows by demonstrating how an AI design-to-code system can learn not only broad design patterns, but also context-specific cues gleaned from repeated user interactions. Over time, the model's predictions become more personalized, enabling more efficient prototyping and smoother creative exploration. This vision for adaptive design assistance advances the field toward intuitive, user-centric methodologies, ultimately bridging the gap between automated code generation and designer-driven innovation.

7 Limitations

In this section, we discuss the limitations of our study concerning benchmarking choices, dataset curation methodology, and their implications for the generalizability of our findings.

7.1 Benchmarking Scope and Model Selection

A potential limitation of this study is the comparison scope concerning state-of-the-art proprietary models (e.g., from OpenAI, Anthropic). Our study prioritized open-weights models to ensure transparency and reproducibility in benchmarking.

Our primary benchmark, FerretUI, was chosen for its strong performance on specialized UI grounding tasks relevant to our design-to-code translation objective, including favorable comparisons against models like GPT-4V in those specific areas [63]. Large proprietary models are typically

generalist architectures, whereas our goal is the specialized translation of visual designs into structured front-end code (HTML/CSS). We hypothesize that the vast scale of these generalist models may exceed the requirements for this focused task. Therefore, using an open-weights, UI-focused model like FerretUI offered a practical, computationally efficient, and openly reproducible baseline for direct comparison and iteration.

This decision implies that while our results demonstrate significant improvements over previous methods and establish strong performance within the open-weights domain for this task, direct claims about performance relative to the absolute state-of-the-art cannot be made based solely on this work. Our focus remains on advancing reproducible research within the specific constraints of design-to-code translation.

7.2 Dataset Curation and External Validity

Our methodology involved significant data cleaning and filtering, particularly the removal of JavaScript and certain web frameworks, which warrants discussion regarding its necessity and impact.

Rationale for Cleaning: The primary aim of our data cleaning process was to generate datasets suitable for training models focused on rapid prototyping where front-end visual fidelity is prioritized. Including JavaScript introduces considerable complexity, as distinguishing between JS for essential interactivity (e.g., menus) and JS for complex logic or animations (often unnecessary for initial prototypes) would require sophisticated filtering mechanisms beyond the scope of this work. Furthermore, removing scripts and sanitizing HTML/CSS significantly reduces the token count, decreasing the memory and computational resources required for training and inference, allowing us to balance dataset quality (focusing on visually static structure) with resource efficiency.

Impact of Filtering: The removal of script nodes and the filtering of websites based on frameworks like React, Gatsby, and Nuxt impacts the dataset's representativeness of the *entire* web. This filtering inherently excludes websites heavily reliant on client-side rendering for their layout and dynamic features. Consequently, our WebUI2Code dataset, while cleaned for structural analysis, may underrepresent complex, dynamic web applications, introducing a potential bias.

External Validity: These curation choices limit the external validity and generalizability of models trained *solely* on this processed data, especially concerning highly dynamic or framework-dependent websites. The claim that the dataset is "structurally representative" should be understood within the context of these filtering steps – it represents a set of real-world UIs amenable to static analysis after cleaning, rather than the full spectrum of modern web development practices. Future work could explore methods to incorporate or dynamically handle JavaScript and framework-specific code to broaden applicability. We also acknowledge the inherent tension between the rendered HTML output we capture and the source code (often using toolkits) a developer might write; our current approach focuses on the former.

8 Conclusion

Our research contributes to advancing web development through several contributions. The introduction of a refined WebUI2Code dataset, especially its collection pipeline, provides a strong foundation for mock-ups to code solutions, while our synthetic datasets enable the exploration of sketch-to-code systems that lower the entry barrier for individuals without technical expertise. Our evaluation of two recent transformer-based models demonstrates improvement on traditional benchmarks and effective results on our synthetic data, aligning with current AI trends and suggesting a future where complex and varied web designs can be translated into functional websites with greater precision. Looking ahead, the focus will be on collecting even larger datasets and large-scale training followed by rigorous experimentation with designers and real users. We envision systems

that can learn and adapt to individual design styles and workflows. These tools could leverage techniques like reinforcement learning to adjust to user feedback and transfer learning to handle diverse design scenarios, creating more personalized and responsive design experiences. By aligning these technological advances with practical workflows and requirements, we will enable the creation of accurate and user-oriented design augmentation tools. Our contributions aim towards a web development aligned with human-centric principles of design and creativity, with the final goal of augmenting human creativity and productivity through AI systems.

References

[1] 2023. Majestic Million. Available at: https://majestic.com/reports/majestic-million.

[2] Nguyen Tuan Anh and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with REMAUI. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 248–259.

[3] B. Aşıroğlu, B.R. Mete, E. Yıldız, Y. Nalçakan, A. Sezen, M. Dağtekin, and T. Ensari. 2019. Automatic HTML code generation from mock-up images using machine learning techniques. In *2019 Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science, EBBT*. IEEE, 1–4.

[4] Kim Bada, Sangmin Park, Taeyeon Won, Junyoung Heo, and Bongjae Kim. 2018. Deeplearning based web UI automatic programming. In *Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems*. ACM, 64–65.

[5] M. Bajammal, D. Mazinanian, and A. Mesbah. 2018. Generating reusable web components from mock-ups. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 601–611.

[6] F. Behrang, S.P. Reiss, and A. Orso. 2018. GUIFetch: Supporting app design and development through GUI search. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 236–246.

[7] T. Beltramelli. 2018. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 1–6.

[8] T. Beltramelli. 2019. *Hack your design sprint: Wireframes to prototype in under 5 minutes.* https://uizard.io/

[9] Bookmark. 2025. Bookmark AI Website Builder. https://www.bookmark.com/. Accessed: 2025-04-27 (Inferred URL, official site).

[10] Kerry Shih-Ping Chang and Brad A. Myers. 2012. WebCrystal: Understanding and reusing examples in web authoring. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3205–3214.

[11] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI design image to GUI skeleton: A neural machine translator to bootstrap mobile GUI implementation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, New York, NY, USA, 665–676.

[12] S. Chen, L. Fan, T. Su, L. Ma, Y. Liu, and L. Xu. 2019. Automated cross-platform GUI code generation for mobile apps. In *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*. IEEE, 13–16.

[13] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar. 2017. RICO: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on UIST (UIST '17)*. ACM, New York, NY, USA, 845–854. doi:10.1145/3126594.3126651

[14] Y. Deng, A. Kanervisto, J. Ling, and A.M. Rush. 2017. Image-to-markup generation with coarse-to-fine attention. In *International Conference on Machine Learning*. PMLR, 980–989.

[15] Elegant Themes Blog. 2025. Wix ADI Review 2025: Is It Really That Powerful? https://www.elegantthemes.com/blog/business/wix-adi-review. Accessed: 2025-04-27.

[16] K. Ellis, D. Ritchie, A. Solar-Lezama, and J. Tenenbaum. 2018. Learning to infer graphics programs from hand-drawn images. In *Advances in Neural Information Processing Systems*. 6059–6068.

[17] J. Ferreira, A. Restivo, and H.S. Ferreira. 2021. Automatically generating websites from hand-drawn mock-ups. In *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*.

[18] X. Ge. 2019. Android GUI search using hand-drawn sketches. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 141–143.

[19] D. Gibson, K. Punera, and A. Tomkins. 2005. The volume and evolution of web page templates. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*. ACM, 830–839.

[20] A. Halbe and A.R. Joshi. 2015. A novel approach to HTML page creation using neural network. *Procedia Comput. Sci.* 45, 197–204. http://www.sciencedirect.com/science/article/pii/S1877050915003580 International Conference on Advanced Computing Technologies and Applications (ICACTA).

[21] Y. Han, J. He, and Q. Dong. 2018. CSSSketch2Code: An automatic method to generate web pages with CSS style. In *Proceedings of the 2nd International Conference on Advances in Artificial Intelligence* (Spain). 29–35.

[22] R. Hartson and P.S. Pyla. 2019. *The UX Book: Process and Guidelines for Ensuring a Quality User Experience* (2 ed.). Morgan Kaufmann.
[23] Y. Hashimoto and T. Igarashi. 2015. Retrieving web page layouts using sketches to support example-based web design. In *SBM*. 155–164.
[24] Saad Hassan, Manan Arya, Ujjwal Bhardwaj, and Silica Kole. 2018. Extraction and classification of user interface components from an image. *Int. J. Pure Appl. Math.* 118, 24 (2018), 1–16.
[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[26] HeyLeia. 2025. Leia AI Website Builder. https://heyleia.com/. Accessed: 2025-04-27 (URL from StackSocial deal).
[27] T.K. Ho. 1995. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, Vol. 1. IEEE, 278–282.
[28] Hostinger. 2025. Hostinger Website Builder (formerly Zyro). https://www.hostinger.com/website-builder. Accessed: 2025-04-27 (Reflecting Zyro's integration).
[29] F. Huang, J.F. Canny, and J. Nichols. 2019. Swire: Sketch-based user interface retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (USA). ACM, 1–10.
[30] G.J. James. 2010. *The Elements of User Experience: User-Centered Design for the Web and beyond* (2 ed.). Pearson Education.
[31] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences* 114, 13 (2017), 3521–3526.
[32] A. Kumar. 2018. *Automated front-end development using deep learning.* https://blog.insightdatascience.com/automated-front-end-development-usingdeep-learning-3169dd086e82
[33] Kenton Lee, Mandar Joshi, Iulia Turc, Hexiang Hu, Fangyu Liu, Julian Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. 2023. Pix2Struct: Screenshot Parsing as Pretraining for Visual Language Understanding. arXiv:2210.03347 [cs.CL]
[34] Y. Liu, Q. Hu, and K. Shu. 2018. Improving pix2code based bi-directional LSTM. In *2018 IEEE International Conference on Automation, Electronics and Electrical Engineering (AUTEEE)*. IEEE, 220–223.
[35] Matchbox Design Group. 2023. The Mechanics Of AI Website Builders: How Do They Work? https://matchboxdesigngroup.com/blog/the-mechanics-of-ai-website-builders-how-do-they-work/. Accessed: 2025-04-27.
[36] T. Memmel and H. Reiterer. 2009. Support Collaboration, Model-based and prototyping-driven user interface specification to and creativity. *Int. J. Univ. Comput. Sci.* 14, 19 (2009), 3217–3235.
[37] Microsoft AI labs. 2019. Sketch2Code. https://sketch2code.azurewebsites.net/
[38] K.P. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Trans. Softw. Eng.* (2018).
[39] S. Natarajan and C. Csallner. 2018. P2A: A tool for converting pixels to animated mobile application user interfaces. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 224–235.
[40] Neo. 2025. Zyro AI | Set-up, Pricing, Templates and Review. https://www.neo.space/blog/zyro-ai. Accessed: 2025-04-27.
[41] Pagecloud Blog. 2024. What happened to The Grid? Does the AI site builder still exist? https://www.pagecloud.com/blog/what-happened-to-the-grid. Accessed: 2025-04-27.
[42] Product Hunt. 2025. Bookmark | AI Website Builder Product Information and Latest Updates (2025). https://www.producthunt.com/products/bookmark-ai-website-builder. Accessed: 2025-04-27.
[43] Product Hunt. 2025. Firedrop Product Information and Latest Updates (2025). https://www.producthunt.com/posts/firedrop-2. Accessed: 2025-04-27.
[44] Product Hunt. 2025. Leia: Website Builder Product Information and Latest Updates (2025). https://www.producthunt.com/products/leia-website-builder. Accessed: 2025-04-27.
[45] Ó.S. Ramón, J.S. Cuadrado, J.G. Molina, and J. Vanderdonckt. 2016. A layout inference algorithm for graphical user interfaces. *Inf. Softw. Technol.* 70 (2016), 155–175.
[46] S. Ren, K. He, R. Girshick, and J. Sun. 2015. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*. 28.
[47] J.M. Rivero, G. Rossi, J. Grigera, J. Burella, E.R. Luna, and S. Gordillo. 2010. From mock-ups to user interface models: An extensible model driven approach. In *International Conference on Web Engineering* (Berlin, Heidelberg). Springer, 13–24.
[48] Huang Ruozi, Yonghao Long, and Xiangping Chen. 2016. Automatically generating web page from a mock-up. In *International Conference on Software Engineering & Knowledge Engineering* (USA). 589–594.
[49] Vinoth Pandian Sermuga Pandian, Sarah Suleri, and Matthias Jarke. 2021. SynZ: Enhanced Synthetic Dataset for Training UI Element Detectors. In *26th International Conference on Intelligent User Interfaces - Companion* (College Station, TX, USA) *(IUI '21 Companion)*. Association for Computing Machinery, New York, NY, USA, 67–69. doi:10.

1145/3397482.3450725
[50] N. Sinha and R. Karim. 2013. Compiling mock-ups to flexible UIs. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 312–322.
[51] Smart Tools AI. 2024. Bookmark AiDA - AI-Powered Website Builder. https://www.smart-tools.ai/product/BookmarkAiDA. Accessed: 2025-04-27.
[52] Andrés Soto, Héctor Mora, and Jaime A. Riascos. 2022. Web Generator: An open-source software for synthetic web-based user interface dataset generation. *SoftwareX* 17 (2022), 100985. doi:10.1016/j.softx.2022.100985
[53] StackSocial. 2025. Leia AI Website Builder: Lifetime Subscription (Business LITE). https://www.stacksocial.com/sales/leia-business-lite-lifetime-subscription. Accessed: 2025-04-27.
[54] S. Suleri, V.P. Sermuga Pandian, S. Shishkovets, and M. Jarke. 2019. Eve: A sketch-based software prototyping workbench. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–6.
[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
[56] WebsitePlanet. 2025. Recensione di Zyro 2025: può competere con Wix? https://www.websiteplanet.com/it/website-builders/zyro/. Accessed: 2025-04-27 (Notes Zyro's integration into Hostinger).
[57] Wix Blog. 2024. Wix ADI: How Design AI Elevates Website Creation for Everyone. https://www.wix.com/blog/wix-artificial-design-intelligence. Accessed: 2025-04-27.
[58] Wix.com. 2025. Creare sito web con intelligenza artificiale | AI per creare siti web. https://it.wix.com/ai-website-builder. Accessed: 2025-04-27.
[59] WowSlider. 2024. Segnalibro Costruttore di Siti Web AI - Recensioni, Tutorial, Alternative. https://wowslider.com/ai-builder/it/bookmark.html. Accessed: 2025-04-27.
[60] WowSlider. 2024. Zyro AI Costruttore di Siti Web - Recensioni, Tutorial, Alternative. https://wowslider.com/ai-builder/it/zyro.html. Accessed: 2025-04-27.
[61] Jason Wu, Siyan Wang, Siman Shen, Yi-Hao Peng, Jeffrey Nichols, and Jeffrey P Bigham. 2023. WebUI: A Dataset for Enhancing Visual UI Understanding with Web Semantics *(CHI '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3544548.3581158
[62] Y. Xu, L. Bo, X. Sun, B. Li, J. Jiang, and W. Zhou. 2021. image2emmet: Automatic code generation from web user interface image. *J. Softw.: Evol. Process* 33, 8 (2021), e2369.
[63] Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. 2024. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In *European Conference on Computer Vision*. Springer, 240–255.
[64] Young-Sun Yun, Jinman Jung, Seongbae Eun, Sun-Sup So, and Junyoung Heo. 2019. Detection of GUI elements on sketch images using object detector based on deep neural networks. In *Proceedings of the Sixth International Conference on Green and Human Information Technology* (Singapore). Springer Singapore, 86–90.
[65] Z. Zhu, Z. Xue, and Z. Yuan. 2018. Automatic graphics program generation using attention-based hierarchical decoder. In *Asian Conference on Computer Vision*. Springer, Cham, 181–196.

A Web Scraping Method

Our methodology for acquiring and processing website HTML code involves retrieval using Selenium with Google Chrome, sanitization to remove undesirable code and fix issues, and formatting to eliminate comments and inconsistencies. To obtain CSS code, we locate CSS file links in the HTML, process them with a custom parser built using tinycss2[2], optimize by excluding irrelevant rules and properties, consolidate into a single file, and update HTML references. Screenshots are captured using Selenium. A screenshot classifier filters the images, keeping only those preserving the page structure after HTML/CSS reduction, ensuring dataset quality for downstream machine learning. The following subsections detail our scraping method. The process of obtaining the HTML code of a website involves three distinct stages. Firstly, retrieving the HTML code rendered by the browser, secondly sanitizing the code, and finally, cleansing and formatting the sanitized code appropriately. With the term "sanitize", we refer to the process of removing unnecessary code lines, and also fixing code syntactical errors, like tags not closed or in the wrong position. "Cleansing"

[2]https://pypi.org/project/tinycss2/

and "formatting" include the removal of comments, multiple white spaces or tabs, and adjusting the structure and layout of the code, making it uniform and more readable. This involves fixing the indentation, adopting a consistent use of quotes (single or double quotes), and aligning tag attributes. Additionally, an HTML parser is used to extract statistics such as the number of HTML nodes and the number of different HTML tags and classes.

A.0.1 Retrieve website's HTML code. For the first step, Selenium is used, an open-source automated testing tool commonly used for web application testing. Selenium enables the automation of the process of interacting with a website and retrieving its HTML code. The chosen browser is Google Chrome, with a 1280x1024 window size.

A.0.2 Code sanitizing. To sanitize the code, a sanitize-html tool is used, it is built on top of htmlparser2 and effectively removes undesirable HTML code by eliminating tags specified in a deny list. It also corrects poorly closed tags and allows tag and attribute substitution.

Tags that do not affect website structure, are associated with external resources, or impact only the dynamic behavior of the websites are excluded, like <script>, <meta>, <noscript>, <svg>, <path>, and <iframe>.

Additionally, attribute substitutions are performed for tags including <img>, <href>, <picture>, <a>, <source>, <link>, , and <figure>. Specifically, "data-src" and "data-lazy-src" attributes are replaced with "src", and "data-srcset" and "data-lazy-srcset" are substituted with "srcset". All links to images are also replaced with a link to the default image within the project.

These HTML attributes are normally replaced asynchronously and are used to speed up website rendering and enhance user experience. Since in our scenario this is not needed, and images are substituted with a default one, we can substitute them during this phase. The HTML tags remain unchanged, while the attribute name is replaced according to the substitutions previously listed. The attribute value is left unaltered, except for the link to the default image change.

This enables each website to have a default image that can be used in place of the original images. This resolves the issue of resource downloads for each website. It will also provide a common appearance for images, facilitating their recognition in subsequent machine-learning tasks.

Another transformation used is the replacement of all <ol> tags with <ul> tags to minimize tag variability when there are no apparent structural differences.

A.0.3 Code cleansing and formatting. For the last step, a tool called clean-html is used. It cleans up HTML code, by removing comments, random line breaks, and mixed tabs. It also formats and indents code correctly. The first processing step is responsible for a major reduction in the line number through filters and transformations. In the final step, empty lines are removed. Each non-empty line can potentially generate one or more lines due to formatting, because it distributes one HTML tag per line, with a few exceptions.

A.1 CSS code

The methodology for obtaining CSS code starts with getting all the CSS related to the website. Each of those is then processed individually, cleansed, and minimized. In the end, the CSS files are merged, and their references are updated.

A.1.1 Get the CSS files related to the HTML file. The first step consists of searching for CSS file references in the HTML file obtained in the previous phase. Each file is then downloaded and processed.

A.1.2 CSS file processing. To process each CSS file, a custom-made parser was used. This parser is built on top of tinycss2, which is a low-level CSS parser and generator capable of processing CSS strings and returning CSS tokens and objects.

This allows for identifying all CSS components and recognizing CSS patterns such as qualified rules or at-rules. Each rule, based on the category is decomposed into different parts and recursively analyzed.

A.1.3 CSS file minimization. To minimize CSS code, the general idea is to remove all code that does not impact the website's appearance.

In fact, it is common practice to put all the style rules for all pages of a website inside one or more common CSS files, avoiding code duplication. However, in our scenario, we are interested only in the rules that affect the page rendered by the previously gathered HTML file. This means that, in many cases, CSS files can be reduced by a lot.

An even bigger reduction is possible when references to big CSS files from frameworks or libraries are present. This is because, usually, only a small portion of their classes are used. Some examples of those frameworks are Bootstrap, Tailwind CSS, and Bulma.

The strategy is to exclude the rules specified for tags or classes, which are not used in the HTML file. For this reason, a complete list of all the tags and classes used in the HTML file is extracted.

Moreover, CSS properties that have a small impact on the appearance of the resulting website screenshot are excluded too. These properties include those related to the website's dynamic behavior, and style properties that do not bring structural changes. In addition, browser-specific CSS properties that are valid for other browsers but not for the one used in the experiment are excluded too.

Table 4 shows the lists of excluded properties, divided by type. Only a smaller portion of the Mozilla Firefox and Internet Explorer properties is shown for readability. The full list can be viewed in the code repository.

Experimental results show that the aforementioned measures result in an average size reduction of the number of lines in the output CSS file by a factor of 10.

A.1.4 Merge of CSS files. To simplify matters, a single CSS file is created by combining all the processed CSS files. Any references to CSS files in the HTML code are updated to point to this specific local file.

A.2 Screenshot extraction

To capture website screenshots Selenium is used, with the same setup as when obtaining HTML code. Two possibilities exist: one connects the website URL and captures the screenshot, while the other (the one used in our experiments) loads the local HTML file and captures the screenshot, producing a website image representative of the processed HTML and CSS files.

Another useful feature is added to close the "accept cookies" pop-ups, which are common on many websites and usually occupy a significant portion of the resulting screenshot. This is particularly important in the first scenario, as in the second one, numerous pop-ups are eliminated due to removing during the sanitizing process of the <script> tag that typically contains them. This functionality simply attempts to locate buttons with common words to dismiss the popups, such as "I Accept", "Ok", and other variants and clicks on them.

A.3 Collection of statistics

Various statistics are extracted for each website with the purpose of monitoring certain metrics that hold potential significance for the development of subsequent machine learning models or other relevant tasks. Statistics are saved in JSON files, one per website.

type	properties
dynamic	transition, transition-timing-function, transition-delay, transition-duration, transition-property, animation-delay, animation, animation-direction, animation-duration, animation-fill-mode, animation-iteration-count, animation-name, animation-play-state, animation-timing-function
various	font-style, text-transform, letter-spacing, word-spacing, line-height, text-shadow, box-shadow, background-image, background-repeat, background-position, hyphens, border-radius, border-style, border-color, order-width, -webkit-font-smoothing
Mozilla Firefox	-moz-appearance, -moz-border-right-colors, -moz-binding, -moz-border-bottom-colors, -moz-box-align, -moz-border-left-colors, -moz-box-flex, -moz-border-top-colors, -moz-box-direction, -moz-box-shadow, -moz-box-ordinal-group, -moz-box-orient, ...
Internet Explorer	-ms-accelerator, -ms-behavior, -ms-block-progression, -ms-content-zooming, -ms-filter, -ms-flex, -ms-flex-align, -ms-flex-direction, -ms-flex-item-align, -ms-flex-line-pack, -ms-flex-order, -ms-flex-pack, -ms-flex-wrap, -ms-grid-column, ...

Table 4. List of excluded CSS properties divided by type.

The dimensions of the recorded screenshot, including its width and height, are preserved alongside the count of lines present within the CSS and HTML files.

The number of nodes, CSS URLs, distinct CSS classes, and tags is extracted from the HTML files. Moreover, CSS files provide information about CSS classes and properties, as well as those excluded during minimization.

Statistics are collected also for the "raw" HTML and CSS files, those without the sanitizing and minimization process, to evaluate the impact of such procedures.

A.4 Web Scraping Quality Assessment

We perform a series of three initial experiments to assess the performance of our web scraping method on different types of data. The results of these experiments are then human evaluated and used to train the classifier, which is subsequently utilized in the final data collection process:

A.4.1 Evaluation over blog websites. To validate the process and evaluate the script's behavior, an initial experiment was performed on a limited number of websites. A set of blog websites was identified as suitable for this purpose, given their relative simplicity and standard appearance.

Specifically, a list of 51 popular blog websites was obtained from the website https://passionwp.com/most-popular-blogs/.

Subsequently, each resulting website screenshot was reviewed and compared to the original website's appearance, without processing or minimization. Based on this comparison, a comprehensive list of observations was recorded, considering factors such as the degree of similarity between the processed screenshot and the original website, the identification of website frameworks, and the nature of the differences between the two versions.

The differences that are typical effects of HTML processing, such as image substitution, are not considered part of the abnormal differences.

A grade was given to each website from 0 to 5:

- 5 to websites almost identical to the original, and with minor differences
- 4 to websites similar to the original, with slight differences, or with differences in small parts of the website (ex: a list is different in a part of the footer)
- 3 to websites with a structure comparable to the original, but with some differences
- 2 to websites with large portions of the screenshot that do not reflect the original website, or with major differences
- 1 to empty websites, websites without styles, and websites completely different from the original ones
- 0 to websites with errors, that did not produce a final screenshot.

A.4.2 Framework detector. Based on the analysis of the first experiment, it was observed that some critical results that received a grade of 1 exhibited the presence of a web framework. As a result, additional system functionality was introduced to detect web frameworks. This is achieved by examining certain keywords and attributes in the website's HTML code. The web frameworks that are considered include React, Gatsby, Next, Nuxt, Backbone, Vue, Angular, and Ember. Table 5 shows the keywords searched for each framework.

By comparing the previously assigned grades of websites with the detected frameworks, it was found that only some of these frameworks consistently produced poor results, while three of them (Vue, Angular, Ember) did not. Therefore, if a framework from the remaining five (React, Gatsby, Next, Nuxt, Backbone) is present, the website is marked as "excluded".

A.4.3 Results. Upon removing the "excluded" websites (grade -1), only a few websites had bad results (grades 1, 2). Overall, 62.75% of websites had good results (grades 3, 4, 5).

After calculating the statistics on the good results, we can see some interesting trends, like the average reduction of lines of CSS code by over 90%, and a reduction of HTML lines by more than 35%.

A.4.4 Majestic million list. A second experiment was conducted on a portion of a larger list of websites.

The list used was Majestic Million, a list of a million website domains with the most referred subnets. The initial 100 websites were analyzed in this experiment.

The results of this experiment are worse than the first one, as was expected by introducing all kinds of websites, some more complicated than blogs. In particular, there is a significant increase in the number of websites with errors from 1 to 10, and websites with very low grades (white pages, websites without CSS).

The reason could be that, since these websites are more popular and drive more traffic, they have additional measures to prevent web scraping. In addition, they are more sophisticated and complex overall.

Framework	Keywords
React	data-reactid=”.*?” React.createElement’ ReactDOM.render’
Gatsby	gatsby- _gatsby GATSBY_.*_POST
Next	_app.js _document.js _error.js _documentSetup _appContent __NEXT_DATA__
Nuxt	nuxt- __NUXT__.js fetch__.js nuxt.js
Backbone	backbone– backbone.js backbone.min.js
Vue	vue– Vue.js Vue.min.js
Angular	ng- angular.js angular.min.js
Ember	ember- ember.js ember.min.js

Table 5. Keywords used to detect the presence of the different web frameworks.

Grades	Total
0	1
-1	14
1	0
2	4
3	7
4	9
5	16
total	51

Table 6. Websites grades obtained during first experiment.

Averages	Raw	Processed
css classes	1788.03	143.39
css classes skipped	0	1596.25
css properties	220.31	78.42
css properties skipped	0	17.83
css urls	8.14	8.11
html classes	238.64	234.75
html tags	35.94	26.86
n html nodes	860.75	699.58
n lines css	20240.5	2255
n lines html	1542.25	996.08

Table 7. Websites statistics obtained during first experiment.

Grades	Total
0	10
-1	35
1	29
2	6
3	4
4	9
5	7
total	100

Table 8. Websites grades obtained over Majestic million.

A.4.5 Evaluation over .blog websites from Majestic million list. At this point, the idea was to test the tool on another portion of the Majestic Million list. This was the first 100 websites with the .blog top-level domain. This was done to extract from the same list a sublist of easier websites, more similar to the ones used in the first experiment.

The outcomes demonstrate a marked improvement compared to the second experiment and are more in line with the first. The percentage of good website screenshots (grade 3, 4, 5) is slightly higher (72% versus 64.29%), but also in this case, the number of bad results, not excluded by the system's filters is not negligible (12%).

Grades	Total
0	7
-1	9
1	9
2	3
3	8
4	29
5	35
total	100

Table 9. Websites grades obtained from .blog websites.

A.5 Screenshot classifier

Initial experiments indicated that the system performed better on simpler websites. However, the difficulty in obtaining large lists of simple websites led us to examine the problem from a different perspective.

Since most of the poor results are easily recognizable by a human and present common patterns, such as blank white pages or unstructured pages lacking CSS, the idea was to train a convolutional neural network to classify the results as either "good" or "bad", and filter out the second ones, similarly to the websites excluded during the previous phases by the framework detector and the detection of websites with zero CSS classes.

The dataset on which we trained the classifier is composed of previous experiments' results, which have been manually classified as "good" or "bad" and some of them have been removed since they are less easily identifiable than others. It contains 219 images, of which 112 are "good" and 107 are "bad". The dataset is almost balanced, with the first class containing approximately 51.1%.

75% of the dataset is used for training and validation, while 25% is for testing. The training-to-validation split is also 75:25.

Some samples from the dataset are shown in Figure 8

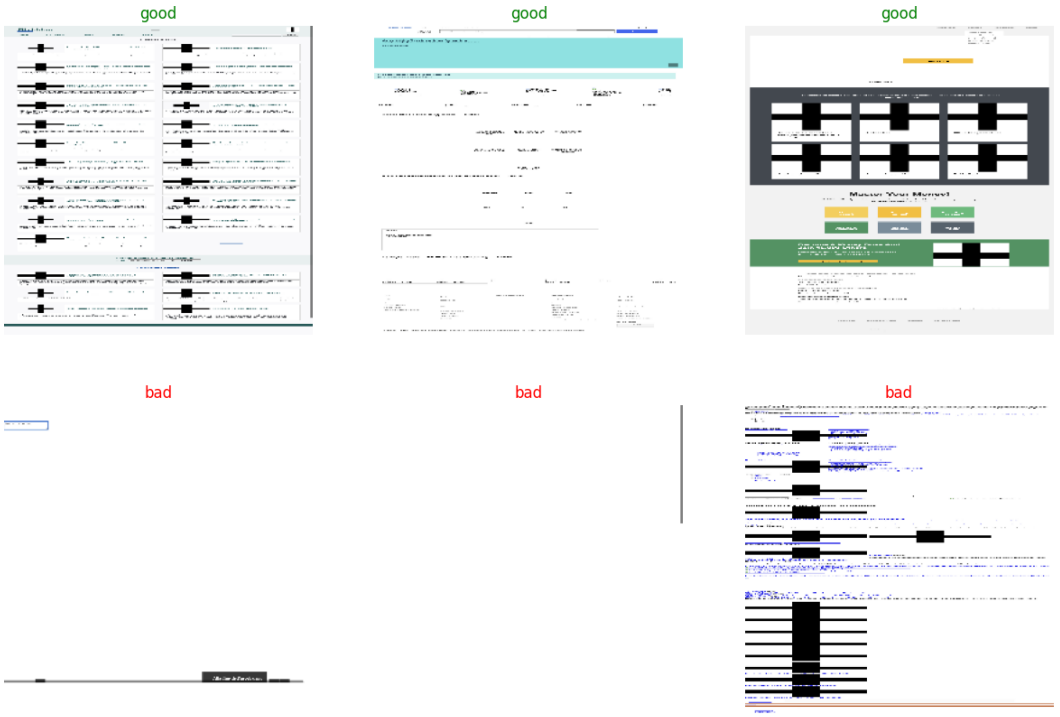


Fig. 8. Samples from the dataset of the screenshot classifier.

A.5.1 Model. The model used is based on the ResNet50 architecture, a widely adopted neural network for image classification.

It is pre-trained on the ImageNet dataset, which contains millions of labeled images across thousands of classes. By leveraging these pre-trained features, the network is able to learn from a small amount of data and achieves high classification accuracy on new images.

The top layer of the ResNet50 model, responsible for the final classification task, is removed. New layers are added on top to fine-tune it for our specific classification task. Additionally, the model includes a dropout layer to reduce overfitting, a random-cropping layer for data augmentation, and layers to resize and scale the images.

The model is trained to classify images into “good” and “bad” using binary cross-entropy loss function and the Adam optimizer.

A.5.2 Training and testing. Several metrics were considered during the training of the model, namely loss, accuracy, precision, recall, and AUC. An “early stopping” strategy was used to avoid overfitting, monitoring the validation loss with a patience value set to 10 epochs. The model was trained for 30 epochs and reached a training accuracy of 83.74% and a validation accuracy of 87.70%.

As a comparison, the model without pre-training on ImageNet reached a training accuracy of 52.03%, and a validation accuracy of 51.22%, always predicting the second class.

This shows the inability of the model to learn from the small data at its disposal. It also shows the impact of transfer learning in a scenario with a scarcity of training data.

During testing, the model reached an accuracy of 81.82%, a precision of 80.00%, a recall of 85.71%, and an AUC of 85.19%. Table 10 and Figure 9 show the classifier results during training, validation, and testing and the confusion matrix on the test set.

	Training	**Validation**	**Testing**
Loss	0.332	0.354	0.703
Accuracy	0.837	0.878	0.818
Precision	0.812	0.864	0.800
Recall	0.889	0.905	0.857
AUC	0.935	0.946	0.852

Table 10. Screenshot classifier performance metrics.

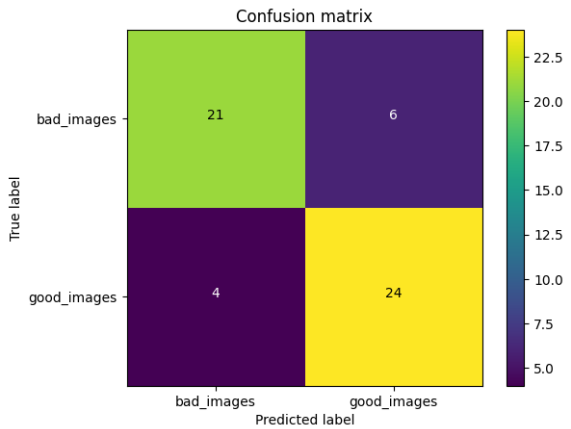


Fig. 9. Confusion matrix illustrating the performance of the screenshot classifier on the test set by displaying the true and predicted classifications.

A new experiment was performed to see the results after the screenshot classifier introduction. The list of websites analyzed comes from the second one hundred websites at the top of the Majestic Million [1] list.

By merging the previous grades 3:5 into the class "good", and the grades 1:2 into the class "bad", it is possible to compare the results with the previous experiments.

	Blogs (1)	MM 1-100 (2)	MM.blog (3)	MM 101-200 (4)
Errors	1	10	7	20
Excluded	14	35	9	26
Bad images	4	35	12	25
Good images	32	20	72	29
TOT	51	100	100	100

Table 11. Table comparing results across different experiments.

Table 11 presents a comparison of the four experiments' outcomes.

The columns of the table correspondingly exhibit the results of the initial experiment performed on blog websites, the second experiment conducted on the first 100 websites listed in the Majestic Million [1] ranking, the third executed on 100 websites enlisted in the Majestic Million ranking with .blog domain, and, finally, the results obtained from the current experiment.

The results of this experiment are similar to the human-classified websites on the top one hundred websites of the Majestic Million list. Specifically, the proportion of websites retained (not excluded and without errors) was 54% (compared to 55% of human evaluation), with a higher proportion being classified as "good", i.e. 53.70% (versus 36.36% of human evaluation).

Again, the number of websites with errors is high, and the motivations are the same ones mentioned in the previous example.

Overall, the final result of this early experiment on a small list is that almost 30% of websites analyzed produce results classified as "good".

A.6 WebUIDataset Collection

The final scraping was conducted on a larger list, containing the top 100000 websites from the Majestic Million list. It was performed on the Politecnico di Torino Big Data Cluster, on a BigDataLab Education environment, with 30 GB of RAM reserved.

It lasted for about 3 weeks, and the 100000 websites were divided into 10 batches of 10000 each. The training of the screenshot classifier took approximately 10 minutes, while the main script ran for around 50 hours for each batch. The other minor scripts consumed a negligible amount of time, while the classification of the non-excluded websites took about 45 minutes for each batch of websites.

The experiment generated about 100GB of files, with the final dataset containing files of websites classified as good taking up 54GB of space.

The results showed similar numbers to the previous experiment in terms of the percentage of the "included" website. The percentage of errors increased from 20% to 29.74%, while the percentage of excluded websites dropped from 26% to 16.46%, and these two experiments somehow balanced the total number of not included websites at around 46%. From the included websites the percentage of them classified as "good" increased from 52.70% to 63.36%.

A.6.1 Statistics. The number of CSS files found in a random website from the analyzed list reproduces a decreasing exponential function, as shown in Figure 10. The average is around 7 files per website, and only less than 10% of the websites have more than 15 CSS files.

The average number of nodes in the processed HTML files is 1061.61%, with an average reduction of 11.34% during cleansing and sanitizing.

Averages	Raw	Processed
css classes	1965.54	139.92
css classes skipped	0	1775
css properties	172.55	67.34
css properties skipped	0	16.56
css urls	7.09	7.07
html classes	224.31	220.69
html tags	34.28	27.44
n html nodes	1197.39	1061.61
n lines css	23037.31	2264.85
n lines HTML	1794.84	1478.43

Table 12. Statistics on the extracted codes from the final experiment.

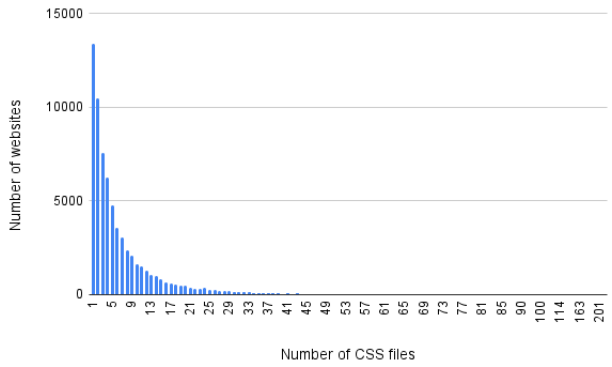


Fig. 10. Distribution illustrating the number of CSS files associated with various websites.

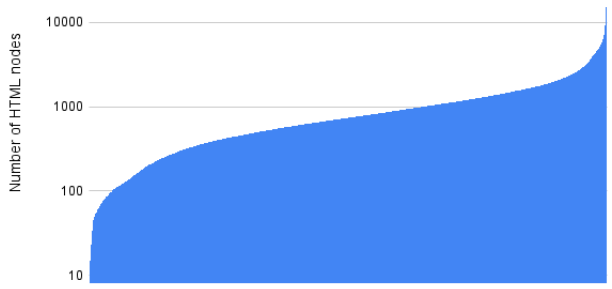


Fig. 11. Distribution illustrating the number of HTML nodes associated with various websites.

The reduction of the number of lines in the CSS files before and after processing is around an order of 10, and it is quite consistent from small files to large files, as shown in Figure 12. The average length before processing is 23037.31 lines, and after processing is 2264.85 lines.

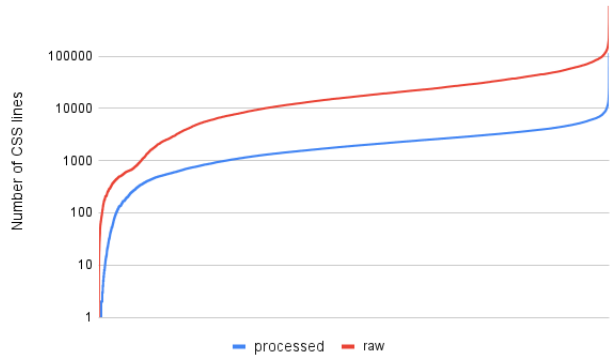


Fig. 12. Comparative distributions (on a logarithmic scale) illustrating the number of lines in CSS files before and after processing, emphasizing the efficiency and impact of the processing step.

A.6.2 Errors. The experiment showed a significant percentage of websites that generated errors, about 29%. Further analysis was done to understand the causes of these errors and their nature.

Table 13 shows all the most common errors encountered during the experiment, with a percentage of occurrence greater than 0.5%.

The majority are related to connection issues or SSL certificates. They occur at the beginning of the experiment, during the connection to the target website to retrieve HTML code. These issues are often caused by firewalls or networking rules that prevent automatic tools from connecting.

The error "List index out of range" occurs during the CSS extraction phase, and is usually caused by using incorrect CSS syntax or invalid characters.

Two errors occurred during screenshot extraction. The first error with the message "Element click intercepted" is raised during the click on Cookies pop-ups, but it is not a blocking error, so the process continues after handling the error.

The second error is "Unable to capture screenshot", which can be due to various issues such as browser incompatibility, network issues, insufficient permissions, or timing issues.

Overall, many of the previous errors are inevitable mainly due to the nature of the experiment setup and the target websites that populate the target list. Some of these websites may be inaccessible to the public, while others may have sophisticated security defenses to avoid suspicious traffic. However, some other errors could be investigated more accurately and handled, like those related to CSS files.

Percentage	Type	Message
16.69%	ConnectTimeoutError	Connection timed out
13.29%	SSLError	<hostname> does not match <allowed hostnames>
13.17%	NewConnectionError	No address associated with hostname
8.14%		List index out of range
7.38%	NewConnectionError	Connection refused
5.29%		Read timed out.
5.05%		Timed out receiving message from renderer
3.92%	SSLError	Self signed certificate
3.80%	SSLError	Certificate has expired
3.79%	SSLError	Unable to get local issuer certificate
3.53%		Element click intercepted
2.90%	NewConnectionError	Temporary failure in name resolution
2.67%	UnknownError	Unable to capture screenshot
2.49%	NewConnectionError	Name or service not known
1.59%	ConnectionResetError	Connection reset by peer
0.98%	OSError	Connection aborted
0.78%	SSLError	Wrong version number
0.69%	SSLError	Alert internal error
0.66%	NewConnectionError	No route to host
0.65%		Remote end closed connection without response
0.50%	SSLError	Alert handshake failure

Table 13. Most common errors encountered during scraping of websites.