ScreenCoder: Advancing Visual-to-Code Generation for Front-End Automation via Modular Multimodal Agents

Anonymous ACL submission

Abstract

Automating the transformation of user interface (UI) designs into front-end code holds significant promise for accelerating software development and democratizing design workflows. While multimodal large language models (MLLMs) can translate images to code, they often fail on complex UIs, struggling to unify visual perception, layout planning, and code synthesis within a single monolithic model, which leads to frequent perception and planning errors. To address this, we propose ScreenCoder, a modular multi-agent framework that decomposes the task into three interpretable stages: grounding, planning, and generation. By assigning these distinct responsibilities to specialized agents, our framework achieves significantly higher robustness and fidelity than end-to-end approaches. Furthermore, ScreenCoder serves as a scalable data engine, enabling us to generate high-quality image-code pairs. We use this data to fine-tune open-source MLLM via a dual-stage pipeline of supervised fine-tuning and reinforcement learning, demonstrating substantial gains in its UI generation capabilities. Extensive experiments demonstrate that our approach achieves state-of-the-art performance in layout accuracy, structural coherence, and code correctness.

1 Introduction

Automating front-end engineering is a critical step toward efficient software development, and recent large language models (LLMs) have advanced the generation of code directly from text instructions (Qwen, 2025; Bolt, 2025). However, this text-based approach faces significant limitations. Generating detailed UIs requires verbose prompts to capture structure and styling, struggles to specify fine-grained visual design like spacing or alignment, and fundamentally deviates from practical design workflows that begin with visual sketches, not paragraphs of text. Relying solely on textual input is therefore sub-optimal for real-world deployment and often fails to capture the full visual intent.

To bridge this gap, multimodal large language models (MLLMs) offer the promise of directly interpreting UI design images and translating them into code (Yang et al., 2023). While conceptually appealing, our analysis reveals that current MLLMs struggle with this task as it requires a unified set of capabilities, visual understanding, structural layout planning, and domain-specific code synthesis, that they are not holistically designed for. Empirically, this leads to two recurring failure modes: (1) perception errors, where components are missed or misclassified, and (2) planning errors, where components are placed incorrectly or violate layout constraints.

To address these limitations, we propose **ScreenCoder**, a modular multi-agent framework that decomposes the UI-to-code task into three interpretable stages: grounding, planning, and generation. The grounding agent leverages a multimodal large language model to localize and semantically label key UI regions. The planning agent then constructs a hierarchical layout tree using domain knowledge of web layout systems. Finally, the generation agent produces HTML and CSS code via adaptive prompt-based synthesis, incorporating both layout context and optional user instructions to support interactive design. This decomposition introduces architectural modularity, enabling more robust component recognition, layout planning, and structured code generation than end-to-end black-box methods. Experiments show that our framework achieves state-of-the-art performance in layout fidelity, structural coherence, and generation quality.

Beyond inference, our framework acts as a **scalable data engine**. This is crucial because training on raw web data is often infeasible, as its length and noise from dependencies and scripts destabi-

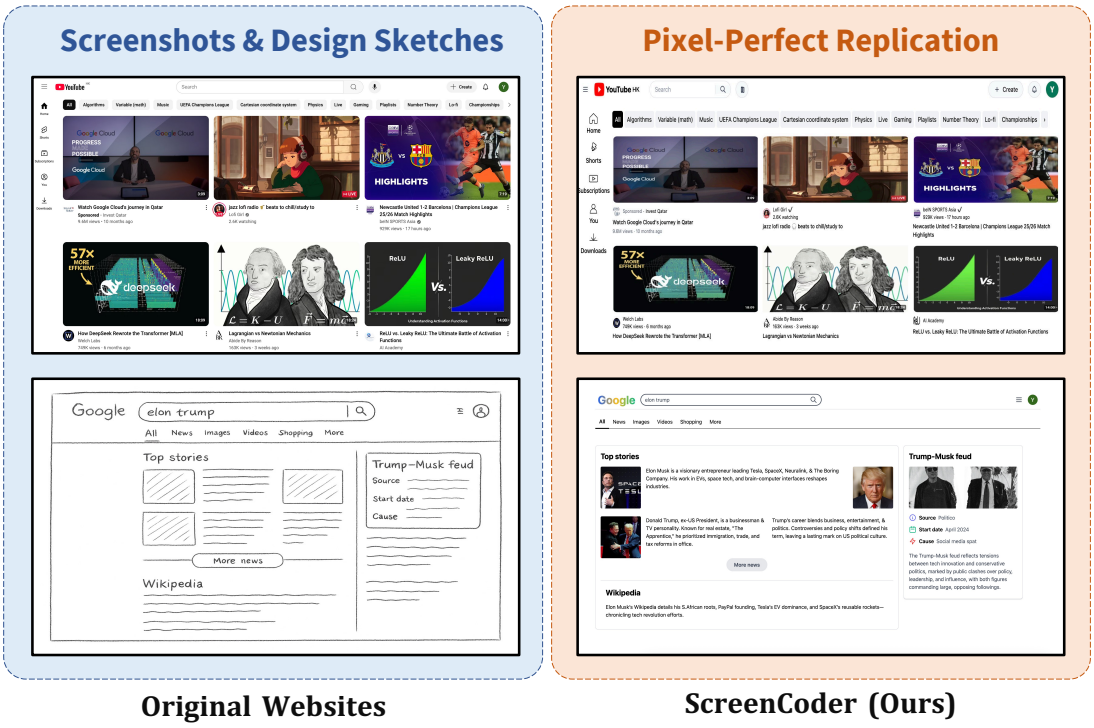


Figure 1: **ScreenCoder accurately transforms website screenshots and design sketches into pixel-perfect front-end code.** The figure showcases a variety of inputs on the left, including high-fidelity screenshots and a low-fidelity design sketch. The right column displays the corresponding webpages rendered from our model's generated code, demonstrating its high-fidelity replication capabilities.

lize training and prevent models from learning the core visual-to-code mapping (Si et al., 2025). To address the challenge, we leverage ScreenCoder to create **Screen-10K**, a new large-scale training dataset of 10,000 high-quality image-code pairs, curated by filtering an initial crawl of 50,000 webpages. We use Screen-10K to significantly enhance open-source MLLM via a dual-stage supervised fine-tuning and reinforcement learning pipeline. Furthermore, to facilitate a more rigorous evaluation of modern models, we introduce **ScreenBench**, a challenging new benchmark composed of 1,000 high-quality, up-to-date websites that reflect contemporary web design. Our framework thus provides a practical path for both scalable dataset creation and robust model alignment. To sum up, our contributions are as follows:

- We conduct a systematic investigation into the limitations of existing MLLMs on UI-to-code tasks and propose a novel modular multi-agent framework that decomposes the complex UI-to-code generation task into three interpretable stages: grounding, planning, and generation, significantly outperforming existing end-to-end multimodal models in layout fidelity and structural coherence.
- Leveraging our framework as a scalable data engine, we introduce Screen-10K, a new large-scale dataset containing 10,000 high-quality image-code pairs, which addresses a critical bottleneck in the field by providing a substantial resource for training more capable UI-to-code generation models.
- To facilitate more rigorous and relevant evaluation, we present ScreenBench, a new challenging benchmark of 1,000 diverse and contemporary web designs. ScreenBench provides a more accurate measure of model performance on real-world tasks compared to existing benchmarks.

2 Related Work

2.1 Multimodal Large Language Models

Multimodal Large Language Models (MLLMs) integrate vision and text to enable joint reasoning. Early models like VisualGPT (Chen et al., 2022) and Frozen (Tsimpoukelli et al., 2021) pioneered this by using pre-trained LLMs to decode

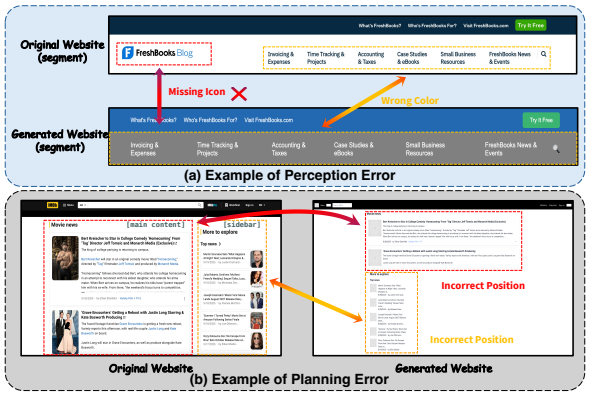


Figure 2: **Analysis of Common MLLM Failure Modes in UI-to-Code Generation.** We identify two primary error categories: (a) Perception Errors, where the model fails to accurately interpret visual details, leading to missing icons or incorrect colors, and (b) Planning Errors, where the model fails to correctly reason about the spatial layout, resulting in elements being placed in the wrong positions.

visual features. Architectural innovations, such as Flamingo's (Alayrac et al., 2022) gated cross-attention and BLIP-2's (Li et al., 2023) Q-Former, further improved vision-language alignment. Modern systems like Gemini 2.5 (Google, 2024) and GPT-4o (OpenAI, 2024) have dramatically scaled these capabilities, excelling at complex multimodal tasks and enabling applications like website generation from images (Zhu et al., 2023). However, despite their impressive general-purpose abilities, these models struggle with domain-specific structured generation, such as UI-to-code synthesis. This limitation stems from a lack of inductive biases for spatial layout and hierarchical planning, as well as a monolithic architecture that hinders the injection of task-specific knowledge.

2.2 Visual-to-Code Generation

Early visual-to-code methods used CNNs and LSTMs to translate UI screenshots into domain-specific languages (DSLs) (Beltramelli, 2018), which offered limited real-world applicability (Xu et al., 2021). Subsequent research shifted towards generating general-purpose HTML/CSS (Chen et al., 2018) and improving component recognition, layout understanding (Cizotto et al., 2023), interaction-awareness (Xiao et al., 2024), and multi-page generation (Wan et al., 2024). Alternative approaches have included OCR-based techniques (Nguyen and Csallner, 2015) and object detection for screen parsing (Wu et al., 2021). More recently, divide-and-conquer methods (Wan et al., 2025; Wu et al., 2025; Gui et al., 2025b,a) have emerged, but often depend on heuristic-based segmentation. Despite these advances, prior works are often brittle, rely on synthetic data, and lack the interpretability needed to jointly model complex visual semantics, layouts, and coding patterns. In contrast, our approach introduces a modular multi-agent framework that decomposes the task into interpretable sub-tasks: grounding, planning, and generation. This allows for explicit reasoning and the integration of domain-specific priors. Our system also functions as a scalable data engine to train future MLLMs, addressing the scarcity of high-quality, realistic image-code datasets.

3 Motivation: Why MLLMs Fail at UI-to-Code Generation?

To motivate our approach, we analyze why state-of-the-art multimodal large language models (MLLMs) struggle with the direct, end-to-end generation of code from UI screenshots. While models like GPT-4o exhibit powerful general visual reasoning, our analysis of their outputs on real-world webpages reveals two primary failure modes, as illustrated in Figure 2: perception failures and planning failures.

First, **perception failures** stem from the model's inability to accurately interpret visual elements. This materializes in two ways: (1) **element omission**, where smaller or less salient components like icons and secondary text are completely missed, and (2) **element distortion**, where an element is recognized but its attributes (e.g., text content, color) are incorrect, or its type is misidentified (e.g., an input field rendered as static text). Second, **planning failures** involve the inability to organize perceived elements into a coherent spatial and hierarchical structure. Even if all components are identified, they are often assembled incorrectly, leading to (1) **element misarrangement**, where components are placed in the wrong positions, and (2) **hierarchical incoherence**, where the model produces a "flat" layout that fails to infer the nested, DOM-like structure essential for modern, responsive web design. This highlights a lack of inductive bias for fundamental front-end layout conventions.

Our analysis indicates these failures arise not from an intrinsic inability to perform any single sub-task, but from the burden of a monolithic, end-to-end approach that overloads a single model with granular perception, complex spatial planning, and

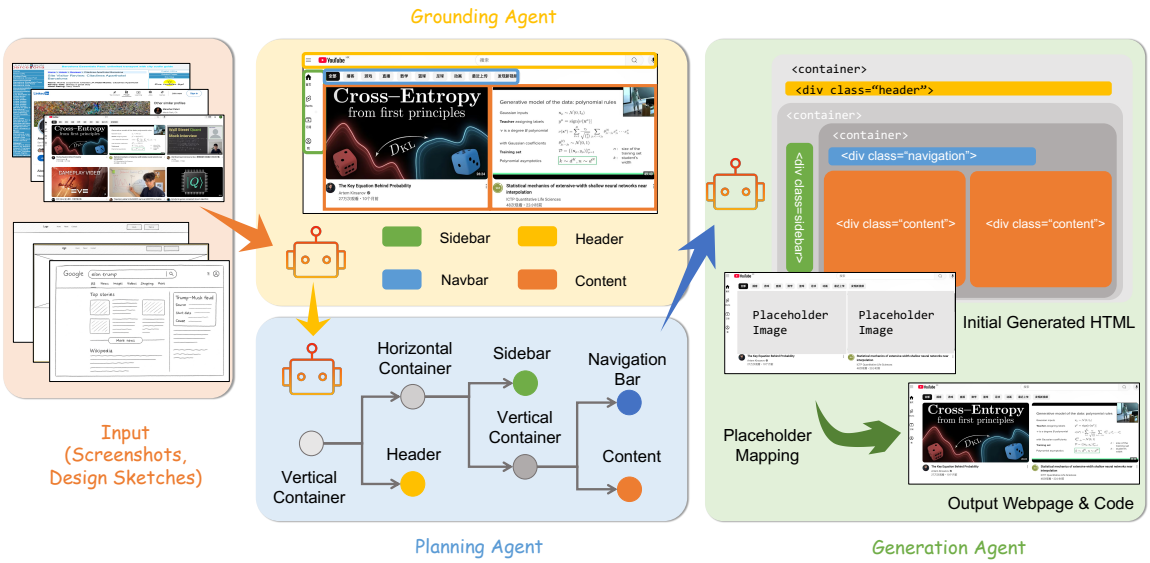


Figure 3: **Overview of ScreenCoder.** Given UI screenshots or design sketches as input, the Grounding Agent first detects and labels key components (*e.g.*, header, navbar, sidebar, content). The Planning Agent organizes these components into a hierarchical layout using front-end engineering priors. The Generation Agent synthesizes initial HTML code with placeholders, followed by content mapping to produce the final webpage and code.

structured code synthesis simultaneously. This task also demands deep domain knowledge of front-end development, such as layout systems and component hierarchies, which general-purpose MLLMs lack. Inspired by agentic workflows that decompose complex problems, we hypothesize that separating these distinct challenges will yield more robust results. This insight directly motivates our modular, multi-agent framework. By explicitly decoupling the task into **grounding** (to address perception failures), **planning** (to address structural failures), and **generation** (to focus on code synthesis), we enable each agent to specialize. Crucially, this modularity allows us to inject specific domain knowledge at each stage, such as established layout conventions in the planning agent, thereby mitigating the failure modes inherent in a single, end-to-end process.

4 Method

We propose a modular, multi-agent framework for UI-to-code generation that decomposes the complex task into three sequential agents: grounding, planning, and generation. This design is explicitly motivated by the failure modes identified in Section 2; each agent is specialized to address a distinct sub-problem, allowing the system to leverage both visual understanding and structured reasoning in a coordinated manner. The grounding agent targets *perception errors* by accurately identifying UI components. The planning agent tackles *planning errors* by constructing a coherent layout hierarchy. Finally, the generation agent translates this structured plan into high-fidelity code. Our overall framework is shown in Figure 3.

4.1 Grounding Agent: Overcoming Perception Errors

The grounding agent serves as the perceptual front-end of our framework, tasked with detecting and semantically labeling major structural components to overcome the *perception errors* (element omission and distortion) common in end-to-end models. This design choice, assigning explicit labels like `sidebar`, `header`, and `navigation`, is crucial for enabling interactive, language-driven design, as it allows users and downstream agents to reference and manipulate specific components via natural language (*e.g.*, "resize the sidebar").

To accomplish this, the agent employs a Multimodal Large Language Model (MLLM) queried with prompts such as "Where is the sidebar?" or "Locate the header area." The MLLM returns a set of grounded regions as bounding boxes and their associated labels:

$$\mathcal{B} = \{(b_i, l_i) \mid l_i \in \mathcal{L}\}_{i=1}^{N}, \qquad (1)$$

where $\mathcal{L} = \{\textit{sidebar}, \textit{header}, \textit{navigation}, \textit{footer}, \textit{main content}\}$. Here, each $b_i = (x_i, y_i, w_i, h_i)$ is a bounding box in pixel coordinates. Unlike tradi-

Algorithm 1 Visual-to-Structural Tree Mapping

Input: Layout dict L, Dimensions W, H
Output: Layout Tree $\mathcal{T}$
$\mathcal{T} \leftarrow$ createNode('root', style={
    'position': 'relative', 'width': '100%', ...})
for label l, box b_l in L **do**
    $N_l \leftarrow$ createNode(l)
    $(x_{\%}, y_{\%}, w_{\%}, h_{\%}) \leftarrow (b_l.x/W, b_l.y/H,$
        $b_l.width/W, b_l.height/H) \times 100$
    N_l.style $\leftarrow$ {'pos': 'abs', 'left': $x_{\%}, \ldots$}
    if l contains subdivisions **then**
        N_l.is_grid_container $\leftarrow$ **true**
    end if
    AddChild($\mathcal{T}, N_l$)
end for
return $\mathcal{T}$

tional object detection, this MLLM-based approach allows grounding to be flexibly guided by text, making the system extensible to new UI concepts.

To ensure robustness, the agent performs several post-processing operations: (1)Deduplication and Conflict Resolution, using class-specific non-maximum suppression (NMS) to filter multiple detections for the same label and retain the most confident one; (2)Fallback Recovery, invoking a heuristic based on spatial priors if a key component is missed (*e.g.*, a wide, short box at the top is likely a header); and (3)Main Content Inference, which robustly defines the primary content area by inferring it as the largest rectangular area not overlapping any detected component. The final output is a layout dictionary which provides the semantic and spatial foundation for the next stage. Unlike traditional object detectors which require costly retraining, our MLLM-based approach is inherently extensible. The system can be adapted to recognize new UI concepts simply by expanding the textual label set $\mathcal{L}$, offering a flexible path for future domain adaptations.

4.2 Planning Agent: Correcting Planning Errors

The Planning Agent mitigates common MLLM failures in spatial reasoning, such as component misarrangement and hierarchical incoherence. Instead of a generative approach, it uses a novel, deterministic Visual-to-Structural Tree Mapping algorithm1 to programmatically translate unstructured visual information into a well-formed layout. The algorithm embeds key domain knowledge from modern web development by converting the flat 2D canvas of bounding boxes into a DOM-like tree, the standard hierarchical data structure for web pages. This deliberately trades the unconstrained flexibility of generative models for structural integrity and predictability, ensuring the generated code faithfully mirrors the source screenshot's layout.

First, our algorithm establishes the global layout by creating tree nodes for primary components (e.g., header, sidebar), converting absolute pixel coordinates into responsive percentage-based values, and using absolute positioning to preserve the macro-structure. It then recursively handles internal component layouts by injecting domain knowledge, designating parent regions with children as CSS Grid containers to arrange nested elements with high fidelity. This process yields an interpretable layout tree that serves as a blueprint, effectively decoupling the abstract planning of the UI structure from the concrete task of code generation.

4.3 Generation Agent: High-Fidelity Code Synthesis

The generation agent translates the hierarchical layout tree $\mathcal{T}$ into executable HTML and CSS. It traverses the tree and, for each node, uses a large language model to generate code based on an *adaptive prompt*. This prompt combines the component's semantic label, its structural context from the tree, and optional user instructions. This approach provides the LLM with rich context, guiding it to produce code that is not only visually correct but also structurally sound and responsive to user intent. The generated code snippets for each component are then assembled according to the tree structure, preserving the hierarchy and layout defined by the planning agent. This closes the loop from visual perception to structured, interactive code synthesis.

4.4 Placeholder Mapping

To restore visual fidelity, we introduce a final placeholder mapping stage that replaces generic image placeholders with their original visual assets. The process begins by using a UI Element Detection (UIED)(Xie et al., 2020) model to extract all visual elements (e.g., icons, images) from the source screenshot. These elements are then partitioned according to the semantic regions defined by the Planning Agent (e.g., header, sidebar).

Within each region, we solve an optimal assignment problem to match the detected elements to the placeholders. We construct a cost matrix based on the negative Complete IoU (CIoU) between the placeholder boxes and the original element boxes, after applying a localized affine transformation to correct for minor rendering discrepancies. This bi-

partite matching problem is solved using the Hungarian Algorithm to find the optimal one-to-one mapping. Finally, the matched image patches are cropped from the original screenshot and inserted into the generated code, restoring the full visual content of the UI.

5 Enhancing MLLMs with Scalable Data Generation and Dual-Stage Post-Training

Beyond its inference capabilities, our framework serves a crucial role as a scalable data engine, addressing a fundamental challenge in training visual-to-code models. Directly training on raw, crawled web data is often infeasible (Si et al., 2025). Real-world code is typically long and noisy, replete with complex dependencies, external links, and irrelevant scripts that make training unstable and hinder the model's ability to learn the core mapping from visual structure to clean, self-contained code.

To overcome this, we leverage our engine to generate Screen-10K, a large-scale dataset of 10,000 high-quality image-code pairs. This dataset was curated by initially crawling 50,000 webpages and applying a rigorous, automated filtering process to retain only the most valuable, well-structured examples. This clean dataset provides the stable foundation necessary for our dual-stage post-training pipeline. First, we perform supervised fine-tuning (SFT) on a 9,000-pair subset to align the model's visual understanding with correct code syntax, establishing a strong baseline. The remaining 1,000 pairs are then used in a subsequent reinforcement learning (RL) stage to further optimize for high visual fidelity. More details about the dataset are shown in Appendix E.

The reinforcement learning stage is based on Group Relative Policy Optimization (GRPO) (Shao et al., 2024). We optimize the policy π_θ to maximize the expected reward over our RL dataset:

$$\max_{\theta} \ \mathbb{E}_{(x,y)\sim\pi_\theta} \left[\mathcal{R}(x,y)\right], \tag{2}$$

where x is the input image and y is the generated code.

To directly optimize for visual fidelity, our reward function $\mathcal{R}(x,y)$ is based on the pixel-level similarity between the original screenshot and the webpage rendered from the generated code. For each output y, we first render it to produce an image, $\text{Render}(y)$. The reward is then defined as the negative Mean Squared Error (MSE) between the original image x and the rendered output. Since RL seeks to maximize reward, using a negative error term incentivizes the policy to minimize the pixel-wise difference:

$$\mathcal{R}(x,y) = -\text{MSE}(x, \text{Render}(y)) \tag{3}$$

where the MSE between two images I_1 and I_2 of height H and width W is calculated as:

$$\text{MSE}(I_1, I_2) = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} \|I_1(i,j) - I_2(i,j)\|_2^2 \tag{4}$$

This reward function provides a strong, holistic signal that guides the model toward generating code that achieves a pixel-perfect visual replication of the original design.

6 Experiments

We evaluate our framework from two complementary perspectives: (1) the visual fidelity and semantic consistency of the generated webpages, and (2) its effectiveness as a scalable data engine for fine-tuning multimodal large language models (MLLMs). We assess the quality of the generated code by comparing the rendered output against ground-truth screenshots using the metrics and benchmarks detailed below.

6.1 Experimental Setup

Datasets. To rigorously evaluate modern UI-to-code models, we introduce **ScreenBench**, a new benchmark of 1,000 high-quality image-code pairs. This benchmark addresses the limitations of existing datasets like Design2Code (Si et al., 2025), which, with its 484 samples from older websites, primarily tests for textual content reproduction over structural complexity. In contrast, ScreenBench is substantially larger and sourced from contemporary web applications, featuring the complex, nested layouts (e.g., CSS Grid/Flexbox) that define modern web design. More details about the benchmark are shown in Appendix E. We also adopt Design2Code (Si et al., 2025) for evaluation.

Evaluation Metrics. We follow the methodology of Design2Code (Si et al., 2025) and evaluate visual similarity using both high-level and low-level metrics. For high-level assessment, we compute the CLIP similarity (Radford et al., 2021) between the rendered output and the reference screenshot. For low-level evaluation, we extract OCR-based visual blocks from both images, align them using text

Model	ScreenBench					Design2Code				
	Block	Text	Position	Color	CLIP	Block	Text	Position	Color	CLIP
GPT-4o	0.745	0.835	0.725	0.702	0.775	0.845	0.962	0.903	0.881	0.917
GPT-4V	0.721	0.815	0.701	0.682	0.758	0.831	0.955	0.895	0.872	0.905
Gemini-2.5-Pro	0.741	0.825	0.752	0.695	0.788	0.841	0.969	0.901	0.879	0.908
LLaVA 1.6-7B	0.635	0.830	0.544	0.592	0.727	0.736	0.910	0.729	0.816	0.802
DeepSeek-VL-7B	0.680	0.773	0.570	0.614	0.732	0.718	0.824	0.702	0.720	0.843
Qwen2.5-VL	0.723	0.828	0.613	0.632	0.762	0.822	0.951	0.815	0.831	0.893
Seed1.5-VL	0.727	0.852	0.742	0.729	0.783	0.829	0.968	0.915	0.897	0.911
DCGen	0.731	0.831	0.713	0.699	0.767	0.836	0.958	0.885	0.865	0.901
LaTCoder	0.735	0.834	0.718	0.703	0.772	0.839	0.960	0.890	0.869	0.904
LayoutCoder	0.733	0.836	0.717	0.706	0.762	0.841	0.965	0.891	0.876	0.909
Websight-8B	0.678	0.768	0.554	0.606	0.748	0.755	0.903	0.767	0.785	0.859
ScreenCoder (Agentic)	**0.768**	**0.857**	**0.755**	**0.734**	**0.812**	**0.865**	**0.975**	**0.925**	**0.908**	**0.922**
ScreenCoder (Finetuned)	0.758	0.841	0.742	0.718	0.791	0.849	0.968	0.913	0.886	0.915

Table 1: **Evaluation results on the ScreenBench and Design2Code benchmarks.** For each metric, the best result is in **bold** and the second best is underlined.

similarity, and then measure four key aspects based on these matched elements: block reproduction accuracy, textual consistency, spatial alignment, and color similarity.

Baselines. We benchmark our approach against a comprehensive suite of state-of-the-art models. This includes leading proprietary MLLMs (GPT-4o (OpenAI, 2024), GPT-4V (OpenAI, 2023), and Gemini-2.5-Pro (Google, 2024)); a range of open-source MLLMs (LLaVA 1.6-7B (Liu et al., 2023), DeepSeek-VL-7B (Lu et al., 2024), Qwen2.5-VL (Bai et al., 2025), and Seed1.5-VL (Guo et al., 2025)); and specialized UI-to-code methods (DCGen (Wan et al., 2025), Websight-8B (Laurençon et al., 2024)), LaTCoder (Gui et al., 2025a) and LayoutCoder (Wu et al., 2025). Our method is implemented on the open-source Qwen2.5-VL-32B model. We show other implementation details in Appendix D.1 and D.2.

6.2 Main Results

As shown in Table 1, our ScreenCoder framework, in both its agentic and fine-tuned variants, consistently surpasses all open-source baselines. The primary agentic model achieves state-of-the-art performance, outperforming even top proprietary systems on the challenging ScreenBench. Furthermore, our fine-tuned model secures the second-best results on many metrics, and achieve comparable performance with close-source models. These results validate our framework's dual utility as both a high-performing inference system and an effective data engine for significantly enhancing open-source MLLMs. Besides automatic evaluation, we also conduct human expert evaluation, whose setting and result are shown in Appedix A.

6.3 Qualitative Analysis

Figure 4 provides a qualitative comparison that highlights the practical advantages of our method. Leading end-to-end MLLMs, such as Qwen-2.5-VL and GPT-4o, demonstrate significant perception and planning failures. They struggle to replicate the target design, resulting in distorted layouts, incorrect component arrangements, and a general loss of styling information. In stark contrast, ScreenCoder produces a high-fidelity result that closely mirrors the original website's appearance and organization. More qualitative results are shown in Figure 1 and Appendix.

6.4 Ablation Study

Effect of Dual-Stage Training. To isolate the contributions of our training pipeline, we conducted a stage-wise ablation study (Table 2). Starting with the base Qwen2.5-VL model, the application of Supervised Fine-Tuning (SFT) yielded significant improvements, particularly in spatial layout awareness. The subsequent Reinforcement Learning (RL) stage further refined the model's capabilities, providing incremental but crucial boosts to achieve our final results.

Data Quality vs. Authenticity. We further validated the necessity of our synthetic data engine by training the base model on 10,000 raw HTML/screenshot pairs collected from the same

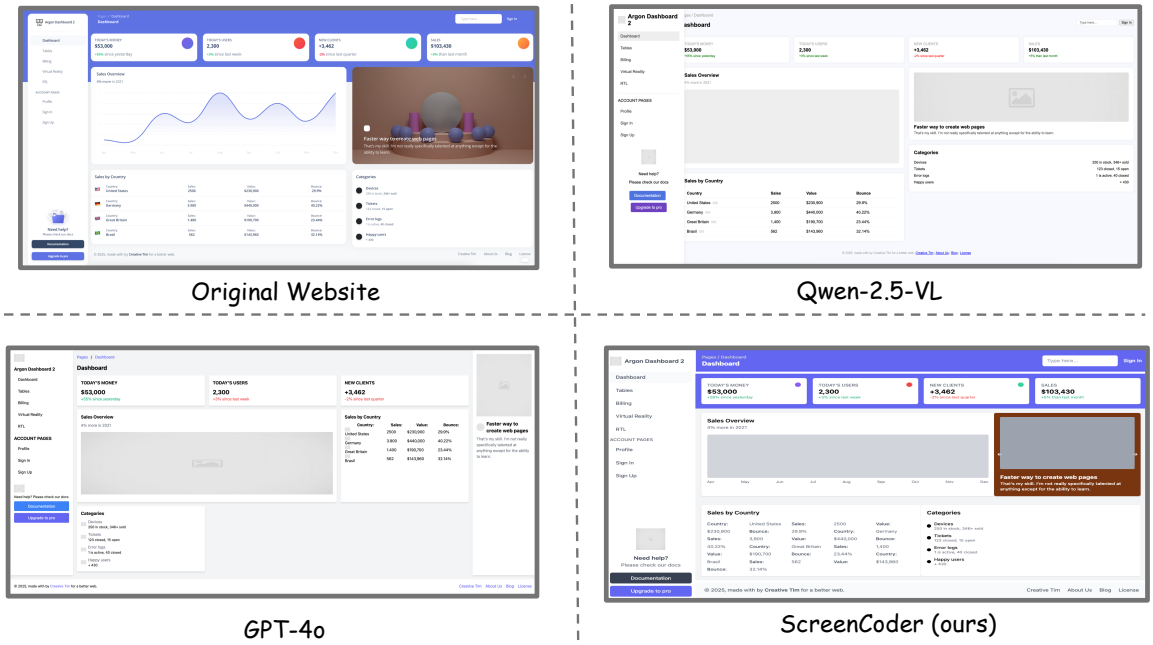


Figure 4: **Qualitative comparison of UI-to-code generation.** While leading MLLMs fail to accurately replicate the target website's layout, styling, and component structure, our method, ScreenCoder, produces a high-fidelity result that closely matches the original design in both appearance and organization.

Training Stage	ScreenBench					Design2Code				
	Block	Text	Pos.	Color	CLIP	Block	Text	Pos.	Color	CLIP
Base Model (Qwen2.5-VL)	0.723	0.828	0.613	0.632	0.762	0.822	0.951	0.815	0.831	0.893
+ SFT (Screen-10K)	0.741	0.835	0.705	0.681	0.784	0.842	0.963	0.890	0.870	0.908
+ RL (Final)	**0.758**	**0.841**	**0.742**	**0.718**	**0.791**	**0.849**	**0.968**	**0.913**	**0.886**	**0.915**

Table 2: **Impact of our dual-stage training pipeline (SFT followed by RL).** SFT provides the foundational structural understanding, while RL refines pixel-level alignment.

Data Source	ScreenBench					Design2Code				
	Block	Text	Pos.	Color	CLIP	Block	Text	Pos.	Color	CLIP
Base Model	0.723	0.828	0.613	0.632	0.762	0.822	0.951	0.815	0.831	0.893
Real-10K (Raw Data)	0.719	0.832	0.605	0.628	0.755	0.825	0.947	0.811	0.825	0.891
Screen-10K (Ours)	**0.741**	**0.835**	**0.705**	**0.681**	**0.784**	**0.842**	**0.963**	**0.890**	**0.870**	**0.908**

Table 3: **Validation of the Data Engine.** Training on raw web data (Real-10K) degrades structural performance due to noise, whereas our synthetic data (Screen-10K) significantly improves it.

websites (Real-10K). As shown in Table 3, training on raw web data resulted in negligible gains or even performance degradation. This confirms that the noise in raw web code such as minified classes and redundant wrappers hinders visual-structural alignment. In contrast, our cleaned, synthetic Screen-10K data provides a far stronger training signal, demonstrating that code quality is more critical than code authenticity. For more ablations regarding separate agent contributions, placeholder mapping and grounding analysis, please refer to Appendix B.

6.5 Scalability and the Cost-Quality Trade-Off.

Our modular framework offers a flexible trade-off between computational cost and output quality via test-time scaling. Users can select smaller, faster models for quick, low-fidelity drafts or larger, more powerful models for high-fidelity, production-ready code. While a high-quality generation takes tens of seconds, this is an acceptable trade-off for its intended use as a workflow accelerator where initial code quality is the priority. This versatility allows the system to be adapted for different stages of the development process. Future work can further enhance this flexibility where developers can compute only to the specific components that require changes.

7 Conclusion

We present ScreenCoder, a modular multi-agent framework for UI-to-code generation that addresses key limitations of end-to-end models, and also functions as a scalable data engine to improve MLLMs via dual-stage post-training. Experiments demonstrate state-of-the-art performance in visual fidelity and code correctness, offering a practical solution and foundation for front-end automation.

Limitations

We discuss two main limitations to our work.

Scope of Dynamic Interactivity. ScreenCoder is currently optimized for the high-fidelity reconstruction of visual layouts and responsive styling, effectively bridging the gap between design and front-end implementation. As such, the system focuses on generating static HTML and CSS. While it successfully handles interactive states (e.g., hover effects, responsive resizing), the automated generation of complex client-side logic, data fetching, or backend integrations remains outside the current scope. Extending the agentic framework to infer and implement intricate behavioral logic represents a promising direction for future work.

Approximation of Non-Standard Geometries. To prioritize the generation of clean, maintainable, and standard-compliant code, our Planning Agent operates within the constraints of the standard web box model. Consequently, when encountering highly artistic or experimental designs featuring non-rectangular geometry (*e.g.*, significant skewing or irregular arbitrary shapes), the system is designed to approximate these elements into a robust grid-based structure. While this design choice ensures that the output remains functional and professionally structured, it may result in a simplification of extreme stylistic deviations in favor of code validity.

References

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, and 1 others. 2022. Flamingo: a visual language model for few-shot learning. *Advances in neural information processing systems*, 35:23716–23736.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, and 8 others. 2025. Qwen2.5-vl technical report. *Preprint*, arXiv:2502.13923.

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

Bolt. 2025. Introduction to bolt. Accessed: 2025-07-23.

C. Chen, T. Su, G. Meng, Z. Xing, and Y. 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*, pages 665–676.

Jun Chen, Han Guo, Kai Yi, Boyang Li, and Mohamed Elhoseiny. 2022. Visualgpt: Data-efficient adaptation of pretrained language models for image captioning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18030–18040.

A. A. J. Cizotto, R. C. T. de Souza, V. C. Mariani, and L. dos Santos Coelho. 2023. Web pages from mockup design based on convolutional neural network and class activation mapping. *Multimedia Tools and Applications*, pages 1–27.

Google. 2024. Gemini api. Accessed: 2024-06-06.

Yi Gui, Zhen Li, Zhongyi Zhang, Guohao Wang, Tianpeng Lv, Gaoyang Jiang, Yi Liu, Dongping Chen, Yao Wan, Hongyu Zhang, Wenbin Jiang, Xuanhua Shi, and Hai Jin. 2025a. Latcoder: Converting webpage design to code with layout-as-thought. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, page 721–732, New York, NY, USA. Association for Computing Machinery.

Yi Gui, Yao Wan, Zhen Li, Zhongyi Zhang, Dongping Chen, Hongyu Zhang, Yi Su, Bohua Chen, Xing Zhou, Wenbin Jiang, and Xiangliang Zhang. 2025b. Uicopilot: Automating ui synthesis via hierarchical code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*, WWW '25, page 1846–1855, New York, NY, USA. Association for Computing Machinery.

Dong Guo, Faming Wu, Feida Zhu, Fuxing Leng, Guang Shi, Haobin Chen, Haoqi Fan, Jian Wang, Jianyu Jiang, Jiawei Wang, Jingji Chen, Jingjia Huang, Kang Lei, Liping Yuan, Lishu Luo, Pengfei Liu, Qinghao Ye, Rui Qian, Shen Yan, and 178 others. 2025. Seed1.5-vl technical report. *Preprint*, arXiv:2505.07062.

Hugo Laurençon, Léo Tronchon, and Victor Sanh. 2024. Unlocking the conversion of web screenshots into html code with the websight dataset. *Preprint*, arXiv:2403.09029.

Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. 2023. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. In *International conference on machine learning*, pages 19730–19742. PMLR.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual instruction tuning. *arXiv preprint arXiv:2304.08485*.

Haoyu Lu, Wen Liu, Bo Zhang, Bingxuan Wang, Kai Dong, Bo Liu, Jingxiang Sun, Tongzheng Ren, Zhuoshu Li, Hao Yang, Yaofeng Sun, Chengqi Deng,

Hanwei Xu, Zhenda Xie, and Chong Ruan. 2024. Deepseek-vl: Towards real-world vision-language understanding. *Preprint*, arXiv:2403.05525.

T. A. Nguyen and C. Csallner. 2015. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 248–259.

OpenAI. 2023. Gpt-4v(ision) system card. Accessed: 2025-07-24.

OpenAI. 2024. Hello gpt-4o. Accessed: 2024-06-06.

Qwen. 2025. Qwen3 blog post. Accessed: 2025-07-23.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning transferable visual models from natural language supervision. *arXiv preprint arXiv:2103.00020*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y.K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2025. Design2Code: Benchmarking multimodal code generation for automated front-end engineering. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3956–3974, Albuquerque, New Mexico. Association for Computational Linguistics.

Maria Tsimpoukelli, Jacob L Menick, Serkan Cabi, SM Eslami, Oriol Vinyals, and Felix Hill. 2021. Multimodal few-shot learning with frozen language models. *Advances in Neural Information Processing Systems*, 34:200–212.

Yuxuan Wan, Yi Dong, Jingyu Xiao, Yintong Huo, Wenxuan Wang, and Michael R. Lyu. 2024. Mrweb: An exploration of generating multi-page resource-aware web code from ui designs. *ArXiv*, abs/2412.15310.

Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael Lyu. 2025. Divide-and-conquer: Generating ui code from screenshots. *Proceedings of the ACM on Software Engineering*, 2(FSE):2099–2122.

Fan Wu, Cuiyun Gao, Shuqing Li, Xin-Cheng Wen, and Qing Liao. 2025. Mllm-based ui2code automation guided by ui layout information. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1123–1145.

J. Wu, X. Zhang, J. Nichols, and J. P. Bigham. 2021. Screen parsing: Towards reverse engineering of ui models from screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 470–483.

Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zhiyao Xu, and Michael R. Lyu. 2024. Interaction2code: How far are we from automatic interactive webpage generation? *ArXiv*, abs/2411.03292.

Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. 2020. Uied: a hybrid tool for gui element detection. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1655–1659, New York, NY, USA. Association for Computing Machinery.

Y. Xu, L. Bo, X. Sun, B. Li, J. Jiang, and W. Zhou. 2021. image2emmet: Automatic code generation from web user interface image. *Journal of Software: Evolution and Process*, 33(8):e2369.

Zhengyuan Yang, Linjie Li, Kevin Lin, Jianfeng Wang, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. 2023. The dawn of lmms: Preliminary explorations with gpt-4v(ision). *ArXiv*, abs/2309.17421.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. 2023. Minigpt-4: Enhancing vision-language understanding with advanced large language models. *ArXiv*, abs/2304.10592.

Appendix

A Human Evaluation

To validate our automatic metrics, we conducted a human evaluation study assessing two critical dimensions: the **perceived visual fidelity** of the generated webpages and the **practical utility** of ScreenCoder in a real-world development workflow.

A.1 Pairwise Comparison of Visual Fidelity

Setup. We recruited six Ph.D. students with web development experience to serve as expert annotators. Following the standard methodology for evaluating generative models, we performed a pairwise comparison study. In each trial, annotators were shown an original UI screenshot alongside two rendered webpages—one generated by **ScreenCoder (Agentic)** and one by a strong baseline (**GPT-4o**). They were asked to vote for the page that was more visually similar to the original ("A is better," "B is better," or "Tie"). To ensure reliable judgments, each comparison was evaluated by three annotators, and a winner was declared based on a majority vote (≥ 2).

Results. The results, summarized in Table 4, reveal a strong human preference for our method. Annotators judged ScreenCoder's output as superior to GPT-4o's in **65%** of cases, while finding it inferior in only **11%**. This outcome confirms that the improvements captured by our automatic metrics translate into a noticeably better perceptual experience, suggesting that our modular approach produces layouts that are more structurally coherent and visually faithful.

Table 4: Pairwise comparison of visual fidelity between ScreenCoder and GPT-4o. Results show the percentage of times each outcome was chosen by human annotators.

Outcome	Win Rate (%)
ScreenCoder Wins	**65%**
GPT-4o Wins (Baseline)	11%
Tie	24%

A.2 Workflow Usefulness Study

Setup. To measure ScreenCoder's impact on developer productivity, we designed a task-based study. Six participants with front-end experience were divided into an **Experimental Group (n=3)**, who used a tool powered by ScreenCoder, and a **Control Group (n=3)**, who used their preferred method (all chose GPT-4o). Each participant was tasked with converting two UI screenshots (one simple, one complex) into functional webpages, with a 20-minute time limit for each task. We measured both the completion time and the quality of the final output, which was rated for UI similarity on a 5-point Likert scale by two independent expert judges.

Table 5: Results of the workflow usefulness study. We compare the performance of developers using ScreenCoder against a control group using GPT-4o.

Metric	ScreenCoder (Experimental)	GPT-4o (Control)
Avg. Completion Time (min)	**8.5**	18.7
Avg. UI Similarity (out of 5.0)	**4.6**	3.4
Tasks Timed Out (out of 6)	**0**	2

Results. As detailed in Table 5, ScreenCoder provides a significant boost to development efficiency. On average, the ScreenCoder group completed tasks **2.2 times faster** than the control group (8.5 minutes vs. 18.7 minutes). Notably, all participants using ScreenCoder finished comfortably within the time limit, whereas two tasks in the control group timed out. Furthermore, the quality of the output was substantially higher for the ScreenCoder group, which achieved an average similarity score of **4.6/5.0**, compared to **3.4/5.0** for the control group (with a strong inter-judge Pearson correlation of 0.78). These findings demonstrate that ScreenCoder acts as a powerful accelerator in a human-in-the-loop process, enabling developers to build UIs faster and with greater accuracy.

B More Ablation Studies and Robustness Analysis

To comprehensively address the individual contributions of our framework's modular components, evaluate functional correctness, and rule out potential training artifacts during the reinforcement learning phase, we conduct the following extended ablation studies and structural analyses.

B.1 Isolation of Agent Contributions

To understand the specific impact of decomposing the visual-to-code task, we ablated the Grounding and Planning agents. We evaluated three configurations on the ScreenBench dataset:

- **w/o Grounding Agent:** The baseline MLLM bypasses explicit component localization, at-

tempting to plan the layout and generate code directly from the raw input image.

- **w/o Planning Agent (Direct MLLM Layout):** The Grounding Agent extracts bounding boxes, but instead of utilizing our deterministic Visual-to-Structural Tree Mapping (Algorithm 1), the MLLM is prompted to infer the HTML DOM hierarchy directly from the flat list of bounding box coordinates.
- **Full ScreenCoder Pipeline:** Our proposed three-stage modular framework.

Table 6: **Ablation of Agent Contributions.** Evaluated on ScreenBench. Removing the deterministic planning agent causes a severe degradation in spatial layout capabilities.

Configuration	Block	Position	Text
w/o Grounding Agent	0.613	0.540	0.810
w/o Planning Agent	0.695	0.632	0.835
Full ScreenCoder	**0.768**	**0.755**	**0.857**

As shown in Table 6, removing the Planning Agent results in a catastrophic drop in the Position score (from 0.755 to 0.632). This confirms our hypothesis that relying on an MLLM to heuristically guess DOM hierarchy from flat bounding boxes is highly error-prone, frequently resulting in collapsed elements and z-index overlap. Our deterministic tree mapping is the primary driver of layout structural integrity.

B.2 Impact of Placeholder Mapping

To isolate the effect of the final visual asset restoration stage, we compared our pipeline with and without the placeholder mapping module. This ablation specifically addresses whether structural layout scores are artificially inflated by the injection of original image patches.

Table 7: **Ablation of Placeholder Mapping.** Evaluated on ScreenBench. Structural scores (Block) remain stable even without original visual assets.

Configuration	CLIP	Color	Block
w/o Placeholder Mapping	0.731	0.645	0.765
Full ScreenCoder	**0.812**	**0.734**	**0.768**

As expected, omitting the placeholder mapping significantly decreases the **CLIP** (0.812 to 0.731) and **Color** (0.734 to 0.645) scores, as original icons, hero images, and complex backgrounds are missing. However, the **Block** score remains effectively stable (0.765 vs. 0.768). This proves that the underlying structural layout generated by the Planning and Generation agents is intrinsically robust and not a byproduct of matching original image patches.

B.3 Quantitative Grounding Accuracy

To ensure the Grounding Agent provides a reliable perceptual foundation for the downstream pipeline, we quantitatively evaluated its localization accuracy. We manually annotated a diverse subset of 200 UI screenshots from ScreenBench with ground-truth bounding boxes for core semantic components: `header`, `sidebar`, `navigation`, `content`, and `footer`.

We report the mean Intersection over Union (mIoU) and mean Average Precision at an IoU threshold of 0.5 (mAP@0.5). The Grounding Agent achieves an **mIoU of 85.4%** and an **mAP@0.5 of 91.2%**. This high localization fidelity confirms that the majority of perception errors (element omission and distortion) are successfully resolved before the planning stage is initiated.

B.4 Functional Correctness and Interactivity

Visual similarity metrics alone are insufficient to guarantee that the generated code is usable in a real-world development environment. We therefore evaluated the functional correctness of the generated HTML.

First, we assessed standards compliance using the W3C Markup Validation Service. ScreenCoder achieved an **88.5% pass rate** for generating structurally valid, error-free HTML5 code, compared to 65.2% for the base Qwen2.5-VL model.

Second, we introduced an **Interactivity Recall** metric. A common failure mode of end-to-end models is hallucinating interactive elements (e.g., buttons, form inputs, links) as static `<div>` or `<span>` tags styled to look clickable. We measured the percentage of truly interactive DOM elements (`<button>`, `<a>`, `<input>`, `<select>`) that were correctly instantiated. ScreenCoder correctly recovered **92.4%** of interactive elements present in the source UI, whereas the end-to-end baseline recovered only 61.8%, demonstrating that our semantic labeling preserves functional interactivity.

B.5 Empirical Analysis of RL Shortcuts and Reward Stability

A known risk in Reinforcement Learning optimization utilizing pixel-based rewards is mode collapse or reward hacking. In the context of UI-to-code generation a model might theoretically learn a shortcut such as wrapping the entire source screenshot in a full-page CSS background image (`background-image: url(...)`) to maximize the pixel-level MSE reward without synthesizing actual structural DOM elements.

We empirically verify that this trivial copy-as-background behavior does not occur in our final model. This stability is inherently maintained by the Group Relative Policy Optimization (GRPO) algorithm, which incorporates a Kullback-Leibler (KL) divergence penalty against the reference SFT model. Because the SFT model is strongly aligned to generate highly structured HTML, any drastic deviation to a flat background-image shortcut incurs a massive KL penalty that outweighs the MSE reward gain.

To quantitatively rule out mode collapse, we analyzed the structural complexity of the generated HTML across the validation set before and after the RL stage.

Table 8: **Structural Integrity Pre- and Post-RL.** The preservation of DOM depth, node count, and text density proves the RL model does not collapse into rendering shortcuts.

Training Stage	Avg. DOM Depth	Avg. HTML Nodes	Text Density
Post-SFT Checkpoint	12.4	145.2	28.5%
Final RL Model	**12.6**	**148.1**	**28.7%**

As shown in Table 8, the average DOM tree depth, the total number of HTML nodes, and the text density (ratio of text characters to markup) remain virtually identical and even slightly improve after reinforcement learning. Furthermore, a direct parsing of the generated CSS revealed a 0% occurrence rate of base64-encoded source images or direct target image URLs used as backgrounds. This confirms that the RL stage successfully fine-tunes CSS properties (e.g., margins, padding, colors) to improve visual fidelity without compromising the structural integrity of the code.

C Additional Qualitative Comparisons

To further demonstrate the effectiveness of our framework, this appendix provides an extended set of qualitative results. The following figures showcase side-by-side comparisons between the webpages generated by our method, ScreenCoder, and those produced by leading baseline models.

These examples were selected to cover a diverse range of website styles and layout complexities. They serve to highlight the common failure modes of existing end-to-end models—such as incorrect spatial arrangements, missing UI elements, and distorted styling—and illustrate how ScreenCoder's modular, agentic approach successfully overcomes these challenges. As shown in the comparisons, our method consistently produces webpages with significantly higher visual fidelity and structural coherence, more closely matching the original design intent of the source screenshots.

D Implementation Detail

D.1 Prompt Template

The prompt templates are shown in Figure 5 and 6.

D.2 Training Details

Our training and experiments were conducted using the Llama Factory codebase on a high-performance cluster of 16 NVIDIA A100 GPUs. We adopted a two-stage training pipeline. For the SFT stage, we fine-tuned the base model on 9,000 samples from our Screen-10K dataset. We adopt the LLaMa Factory (Zheng et al., 2024) as the code base. The model was optimized using the AdamW optimizer with a cosine learning rate schedule, a peak learning rate of 2×10^{-5}, a weight decay of 0.01, and a warmup ratio of 10%. In the subsequent RL stage, we used the remaining 1,000 samples from Screen-10K. We employed the Group Relative Policy Optimization (GRPO) algorithm, continuing from the SFT checkpoint. The reward function was based on the negative Mean Squared Error (MSE) between the rendered output and the ground-truth image, directly optimizing for visual fidelity.

D.3 Extended Visual-to-Structural Tree Mapping

To clarify how bounding boxes are translated into a robust DOM hierarchy, Algorithm 2 details the complete Visual-to-Structural Tree Mapping process. Specifically, it formalizes:

1. **Overlap Resolution:** How the system handles intersecting bounding boxes by determining parent-child containment (nesting) versus spatial conflicts.

2. **Nesting Logic:** A recursive approach to assign smaller components as children of larger encompassing regions.

3. **Grid Specification:** The heuristic injection of `display: grid` or `display: flex` when a parent node contains multiple children, dynamically calculating grid templates based on the children's relative spatial coordinates.

Algorithm 2 Extended Visual-to-Structural Tree Mapping

Input: Layout dictionary $L = \{(b_i, l_i)\}_{i=1}^{N}$, Image Dims W, H, Overlap Threshold τ
Output: Hierarchical Layout Tree $\mathcal{T}$
$\mathcal{T}_{root} \leftarrow$ Node(label='body', style = {'width': '100%', 'min-height': '100vh'})
Sort L descending by bounding box area $(w_i \times h_i)$
▷ *Phase 1: Nesting & Overlap Resolution*
for each (b_i, l_i) in L **do**
  $N_i \leftarrow$ Node(label $= l_i$, box $= b_i$)
  parent_found $\leftarrow$ False
  for each existing node N_p in $\mathcal{T}_{root}$ (bottom-up traversal) **do**
    intersection $\leftarrow$ Area($b_i \cap N_p$.box)
    if intersection/Area(b_i) $> \tau$ **then** ▷ *Child is fully/mostly contained*
      AddChild(N_p, N_i)
      parent_found $\leftarrow$ True
      break
    end if
  end for
  if not parent_found **then**
    AddChild($\mathcal{T}_{root}, N_i$)
  end if
end for
▷ *Phase 2: Responsive Grid/Flex Specification*
for each Node N in $\mathcal{T}$ (post-order traversal) **do**
  if CountChildren(N) > 0 **then**
    $(x_{\%}, y_{\%}, w_{\%}, h_{\%}) \leftarrow$ ConvertToPercentage(N.box, W, H)
    if children are strictly horizontally aligned **then**
      N.style $\leftarrow$ {'display': 'flex', 'flex-direction': 'row'}
    else if children are strictly vertically aligned **then**
      N.style $\leftarrow$ {'display': 'flex', 'flex-direction': 'column'}
    else
      N.style $\leftarrow$ {'display': 'grid'}
      N.style $\leftarrow$ ComputeGridTemplate(N.children) ▷ *Inject CSS Grid domains*
    end if
  end if
end for
return $\mathcal{T}_{root}$

D.4 Inference-Time Configuration and Latency/Cost Trade-offs

To provide transparency regarding the computational requirements of ScreenCoder, we detail the inference-time model deployment and the associated latency and cost trade-offs.

Model Configuration: In our primary experiments, we deploy **Qwen2.5-VL-32B** for the Grounding Agent due to its superior visual perception and spatial grounding capabilities. The Planning Agent executes locally via a deterministic Python script (Algorithm 2), requiring no neural computation. For the Generation Agent, we again utilize **Qwen2.5-VL-32B** (fine-tuned via our SFT+RL pipeline) to synthesize the component-level code, as it seamlessly accepts the cropped component images alongside the layout context prompts.

Prompt Frequency and Parallelism: The pipeline requires exactly **1 prompt per page** for the Grounding Agent to extract the global layout. Subsequently, the Generation Agent issues K **prompts per page**, where K is the number of identified leaf nodes (typically $K \in [4, 6]$ for standard webpages, e.g., header, sidebar, main content, footer). Because the Planning Agent isolates the structural dependencies in advance, these K generation prompts are strictly independent and are executed in **parallel**.

Latency and Cost: Due to parallel execution, the total wall-clock latency is bottlenecked only by the longest single component generation, rather than scaling linearly with page complexity. On our hardware (NVIDIA A100), the end-to-end generation of a complex webpage takes approximately **12 to 18 seconds**.

Financially, while our pipeline issues multiple API calls (if hosted) compared to a single monolithic prompt, the context window for each Generation Agent prompt is drastically smaller. Instead of passing an entire webpage's verbose structural context into one massive output generation (which scales quadratically in attention mechanisms), ScreenCoder isolates generation to bite-sized component chunks. This modularity not only prevents the LLM's attention from degrading but also keeps token usage highly optimized, making the overall inference cost competitive with, or cheaper than, a single massive prompt to a proprietary model like GPT-4o.

E Dataset Transparency, Curation, and Ethics

To address the ethical and legal considerations of training on web-crawled data, and to provide full transparency regarding our data artifacts, this section details the curation pipeline for both the Screen-10K training dataset and the ScreenBench evaluation suite.

E.1 Data Source and Categorization

The initial pool of 50,000 webpages was sourced from an aggregated list of popular, modern websites (e.g., via Tranco and Alexa top lists), prioritizing sites updated within the last two years to ensure the presence of contemporary layout techniques (e.g., CSS Grid, Flexbox).

To ensure the models learn a robust, generalizable mapping rather than overfitting to a specific design paradigm, we stratified the final selected datasets across five primary web design categories. The distribution of these categories for both Screen-10K and ScreenBench is detailed in Table 9.

Table 9: **Category Distribution of Screen-10K and ScreenBench.** The datasets are stratified to ensure broad representation of modern web design paradigms.

Category	Screen-10K (Train)	ScreenBench (Eval)
E-commerce & Retail	2,850 (28.5%)	280 (28.0%)
SaaS & Dashboards	2,400 (24.0%)	250 (25.0%)
Corporate Landing Pages	2,150 (21.5%)	220 (22.0%)
News, Media & Blogs	1,600 (16.0%)	150 (15.0%)
Portfolios & Resumes	1,000 (10.0%)	100 (10.0%)
Total	**10,000**	**1,000**

E.2 Deduplication and Contamination Prevention

A critical requirement for valid evaluation is ensuring no data leakage occurs between the training set (Screen-10K) and the evaluation set (ScreenBench). We implemented a rigorous two-step deduplication process:

1. **URL Deduplication:** We ensured strict disjointness at the highest-level domain (e.g., no pages from `example.com` appear in both splits).

2. **DOM and Visual Similarity Filtering:** Because different domains often use identical underlying templates (e.g., default Shopify or WordPress themes), we computed MinHash signatures on the underlying DOM trees and extracted perceptual hashes (pHash) for the rendered screenshots. Any candidate in ScreenBench with a structural or visual similarity score exceeding 0.85 to any sample in Screen-10K was discarded.

E.3 PII Mitigation and Copyright Handling

Web-crawled data inherently carries risks related to Personally Identifiable Information (PII) and copyrighted visual assets. We applied the following automated mitigations prior to compiling the final datasets:

- **PII Removal:** We utilized an off-the-shelf Named Entity Recognition (NER) model (spaCy) combined with strict regular expressions to detect and mask sensitive text nodes within the DOM. Detected names, phone numbers, email addresses, and physical addresses were replaced with generic placeholder text (e.g., `[REDACTED_EMAIL]`).

- **Copyrighted Asset Substitution:** To prevent the unauthorized redistribution of copyrighted images, illustrations, and proprietary icons, the raw image assets are **not** included in the dataset release. Instead, during the data curation process, all `<img>` tags and CSS `background-image` properties were replaced with visually distinct, solid-color placeholder blocks that match the original asset's dimensions. This preserves the layout and structural integrity required for training while completely eliminating copyright infringement risks.

E.4 Release Artifacts and Licensing

To foster reproducible research while adhering to ethical standards, we will publicly release the following artifacts under the **Creative Commons Attribution-NonCommercial 4.0 International (CC BY-NC 4.0)** license:

1. **Cleaned HTML/CSS Pairs:** The structurally cleaned, PII-redacted, and asset-substituted source code.

2. **Rendered Screenshots:** High-resolution images of the rendered pages (with placeholder images).

3. **Grounding Annotations:** The bounding box coordinates and semantic labels (extracted during our pipeline) to facilitate future research in UI component detection.

Note: The raw, unredacted web crawls and original image assets will be strictly deleted and not distributed.

F Failure Case Analysis and Robustness Check

In this section, we provide a qualitative analysis of failure modes, specifically focusing on "Hard Cases" involving non-standard, artistic layouts that deviate from the standard rectangular box model. We compare the performance of ScreenCoder against the strong end-to-end baseline, Qwen2.5-VL.

As illustrated in Figures 15 to 17, we test the models on a design featuring significant CSS transformations (e.g., `skewY`) and overlapping elements. This analysis highlights a critical trade-off between pixel imitation and structural integrity:

- **Catastrophic Failure in Baselines (Qwen2.5-VL):** Monolithic MLLMs tend to prioritize pixel-level visual matching. When encountering skewed or rotated elements, the baseline attempts to replicate the geometry using absolute positioning and incorrect transform approximations. This results in "spatial hallucination," where elements suffer from Z-index errors, text overlaps, and fragmented coordinate placement. The resulting code, while attempting to look like the input, is functionally unusable.

- **Graceful Degradation in ScreenCoder (Ours):** Our modular approach prioritizes valid DOM structure. The Planning Agent maps the non-rectangular visual input to the nearest robust rectangular grid (CSS Flexbox/Grid). While this results in a loss of the specific artistic nuance (the skew angle is removed), the system fails *gracefully*. The generated code remains clean, responsive, and structurally valid. This demonstrates that ScreenCoder prefers simplification over broken complexity, ensuring the output is always a viable starting point for developers.

G LLM Usage Statement

We used LLMs as an assistive tool for improving grammar and clarity in the text, and for code completion. The core research, including the experimental design and final implementation, was conceived and executed entirely by the authors.

Prompt for Grounding

Seed-1.5-thinking-pro-vision

"""Only return the bounding boxes of the ‘header', ‘footer’, ‘main content’, ‘sidebar’, and ‘navigation’ in this webpage screenshot. Please only return the corresponding bounding boxes in the format of ‘name: <bbox>x1 y1 x2 y2</bbox>’ for each line without extra information and comments!!! Note: 1. All text information, images and other content should be framed, don't miss any information; 2. The areas should not overlap; 3. Output a label and the corresponding bounding box for each line. You can decide whether to include some regions or not."""

Qwen-2.5-VL

"""Only return the bounding boxes of the ‘header’, ‘footer’, ‘main content’, ‘sidebar’, and ‘navigation’ in this webpage screenshot. Please only return the corresponding bounding boxes in the format of ‘header: <bbox>x1 y1 x2 y2</bbox>\nfooter: <bbox>x1 y1 x2 y2</bbox>\nmain content: <bbox>x1 y1 x2 y2</bbox>\nsidebar: <bbox>x1 y1 x2 y2</bbox>\nnavigationr: <bbox>x1 y1 x2 y2</bbox>\n ’for each line without extra information and comments!!! Note: 1. All text information, images and other content should be framed, don‘t miss any information; 2. The areas should not overlap; 3. Output a label and the corresponding bounding box for each line. You can decide whether to include some regions or not. Also, output the image size in the format of "width: <width> height: <height>\n"."""

Figure 5: Prompt Templates.

Prompt for Generation

{"sidebar": f"""This is a screenshot of a container. Here is the user's additional instruction: {user_instruction["sidebar"]}. Please fill in a complete HTML and Tailwind CSS code to accurately reproduce the given container. Please ensure that all block layouts, icon styles, sizes, and text information are consistent with the original screenshot, based on the user's additional conditions. Below is the code template to fill in:

your code here

Only return the code within the and tags.""",

"header": f"""This is a screenshot of a container. Here is the user's additional instruction: {user_instruction["header"]}. Please fill in a complete HTML and Tailwind CSS code to accurately reproduce the given container. Please ensure that all blocks‘ relative positions, layout, text information, and colors within the bounding box are consistent with the original screenshot, based on the user's additional conditions. Below is the code template to fill in:

your code here

Only return the code within the and tags.""",

"navigation": f"""This is a screenshot of a container. Here is the user’s additional instruction: {user_instruction["navigation"]} Please fill in a complete HTML and Tailwind CSS code to accurately reproduce the given container. Please ensure that all blocks‘ relative positions, text layout, and colors within the bounding box are consistent with the original screenshot, based on the user’s additional conditions. Please use the same icons as in the original screenshot. Below is the code template to fill in:

your code here

Only return the code within the and tags.""",

"main content": f"""This is a screenshot of a container. Here is the user’s additional instruction: {user_instruction["main content"]} Please fill in a complete HTML and Tailwind CSS code to accurately reproduce the given container. Please replace the images in the original screenshot with solid gray blocks of the same size; text inside the images does not need to be recognized. Please ensure that all blocks‘ relative positions, layout, text information, and colors within the bounding box are consistent with the original screenshot, based on the user's additional conditions. Below is the code template to fill in:

your code here

Only return the code within the and tags.""”,

"footer ": """This is a screenshot of a container. Here is the user’s additional instruction: {user_instruction["footer"]}. Please fill in a complete HTML and Tailwind CSS code to accurately reproduce the given container. Please ensure that all blocks‘ relative positions, text layout, and colors within the bounding box are consistent with the original screenshot, based on the user’s additional conditions. Please use the same icons as in the original screenshot. Below is the code template to fill in:

your code here

Only return the code within the and tags."""}

user_instruction: dictionary for storing user-defined instructions for each region ("header", "footer", "sidebar", "navigation", "main content").

Figure 6: Prompt Templates.

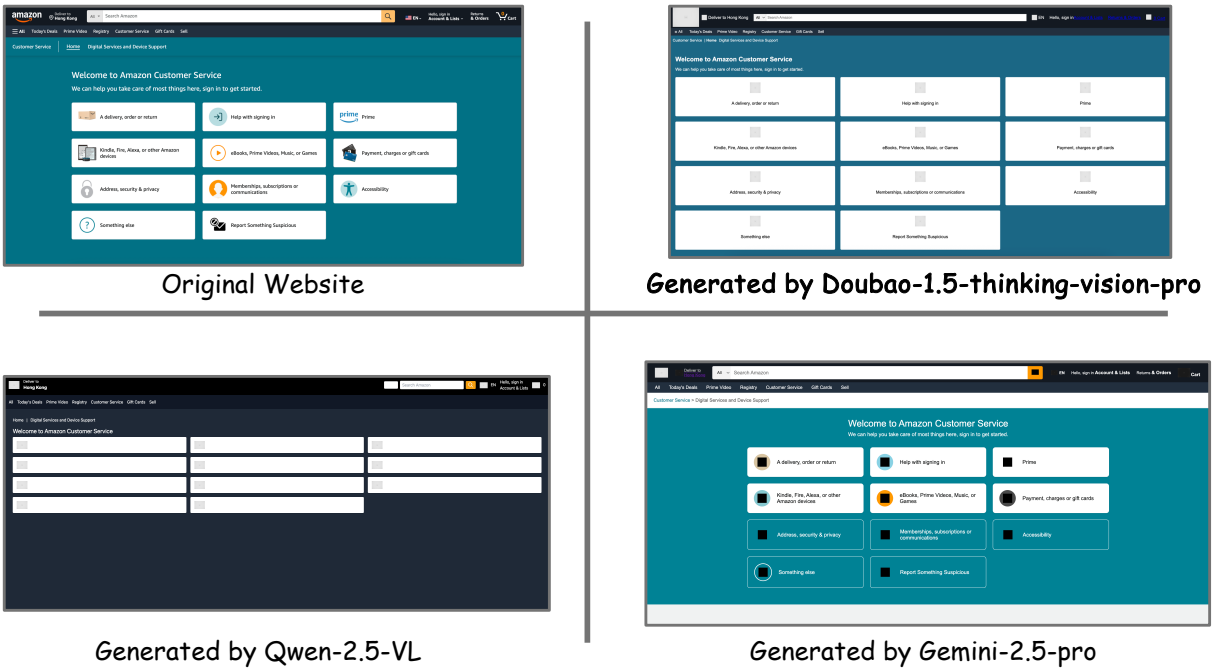


Figure 7: Qualitative comparison between our proposed method and various baselines.

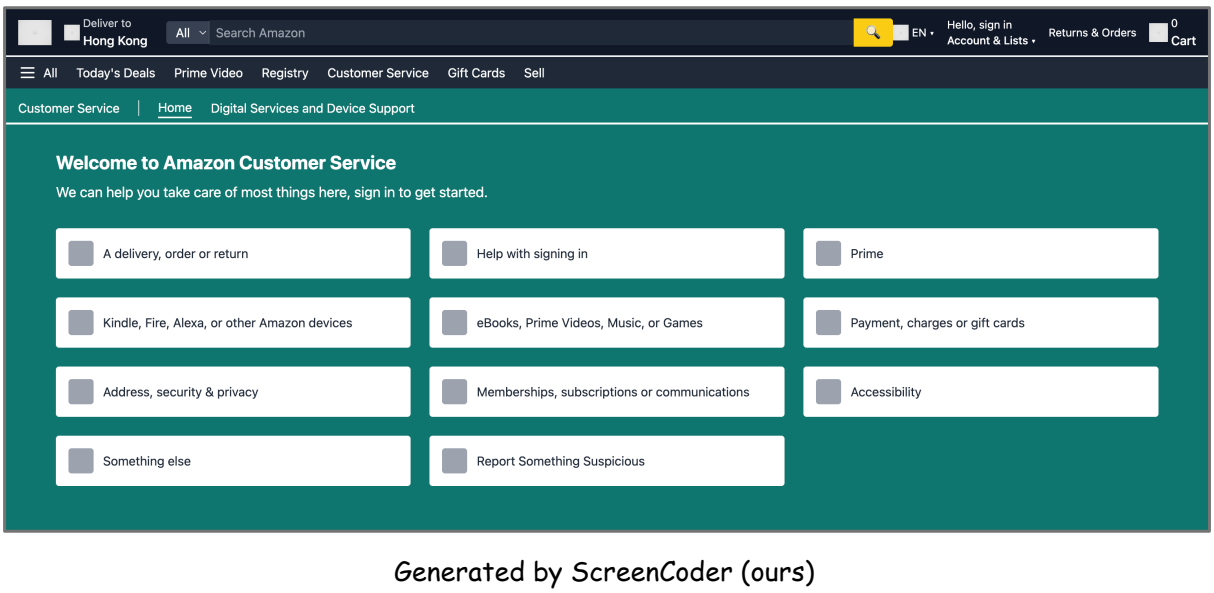


Figure 8: Qualitative comparison between our proposed method and various baselines.

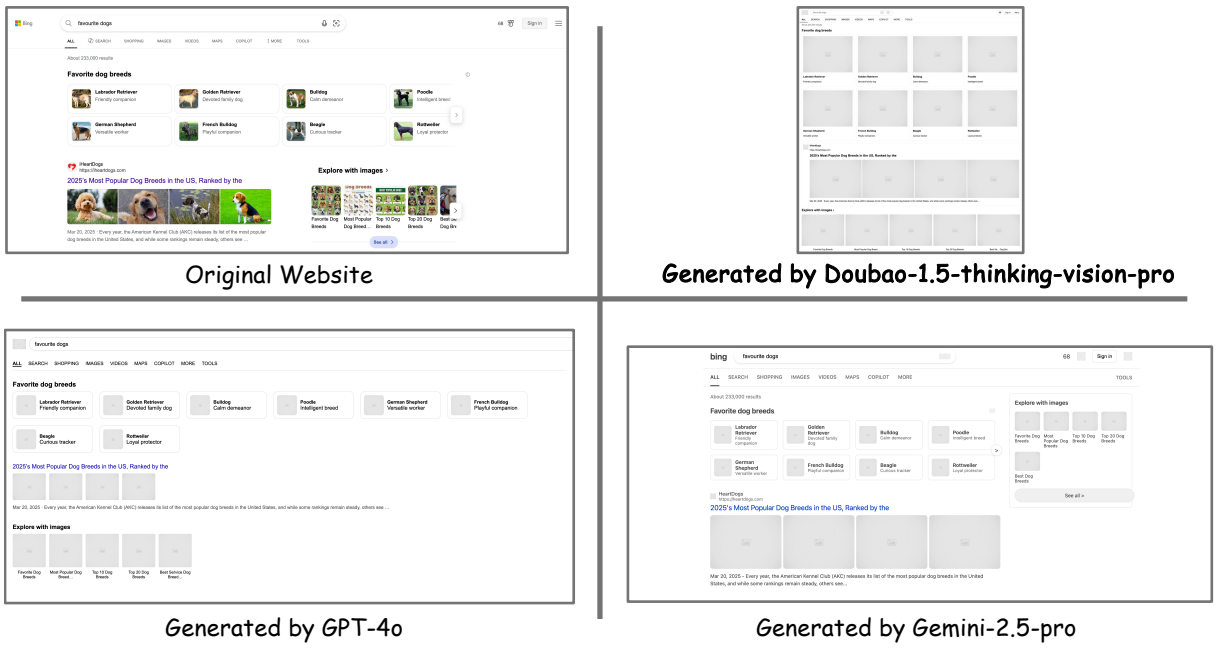
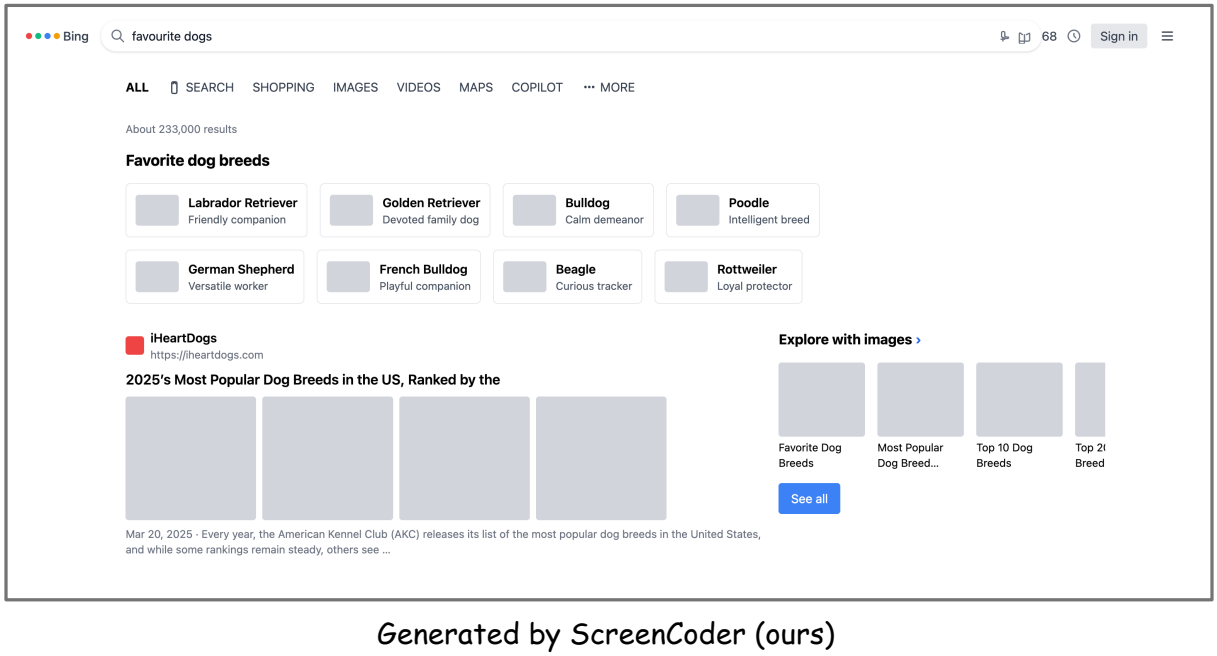


Figure 9: Qualitative comparison between our proposed method and various baselines.



Generated by ScreenCoder (ours)

Figure 10: Qualitative comparison between our proposed method and various baselines.

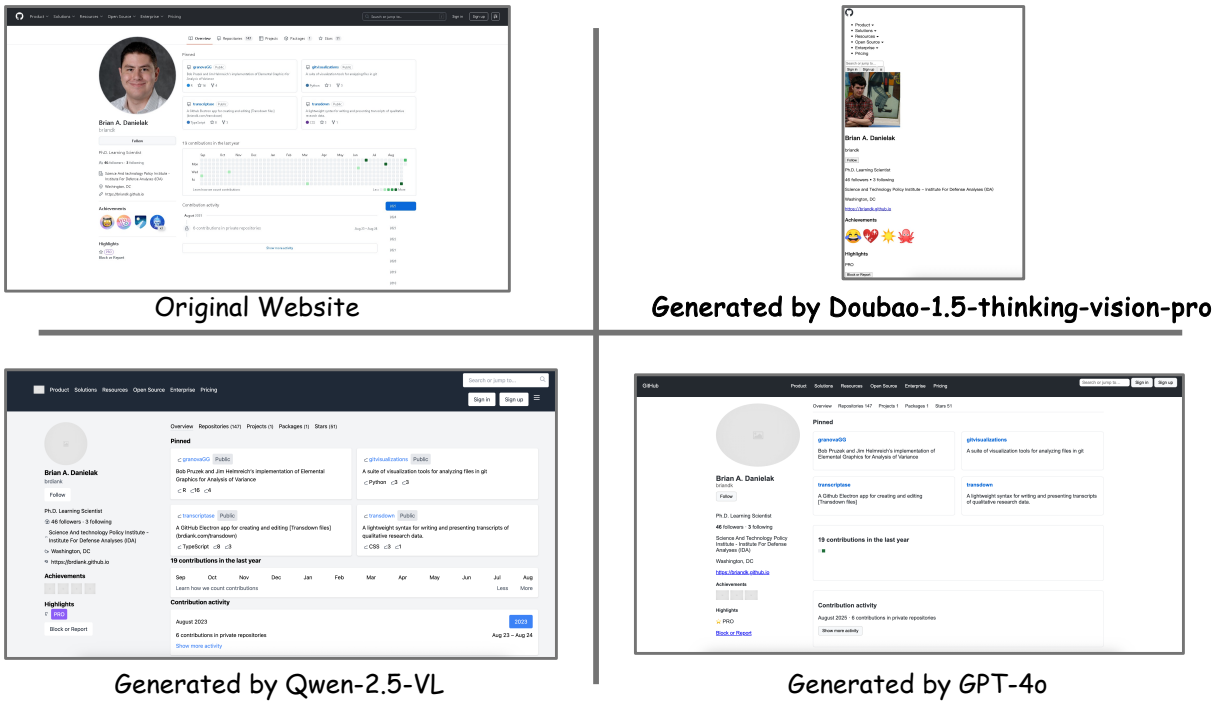
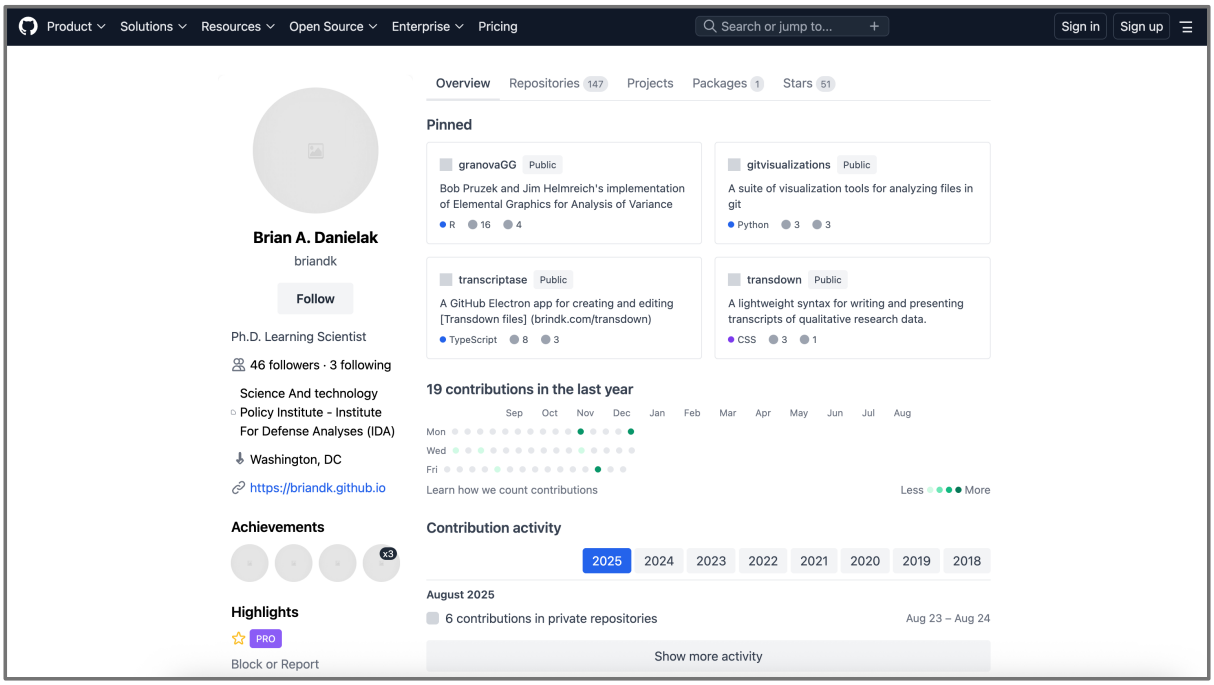


Figure 11: Qualitative comparison between our proposed method and various baselines.



Generated by ScreenCoder (ours)

Figure 12: Qualitative comparison between our proposed method and various baselines.

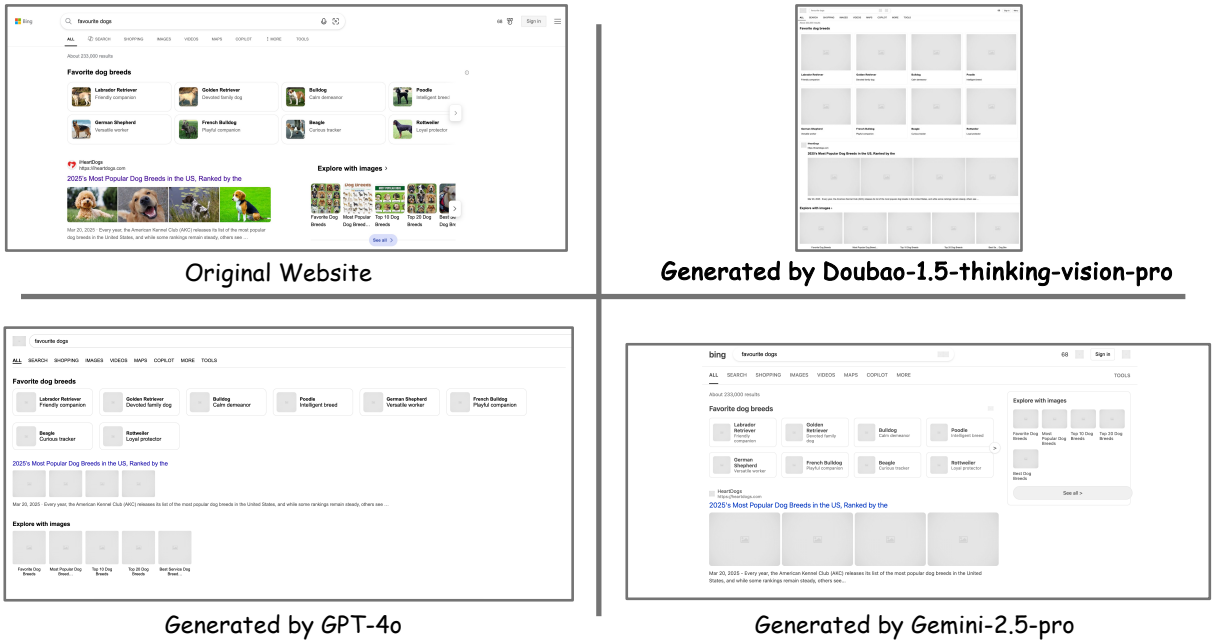


Figure 13: Qualitative comparison between our proposed method and various baselines.

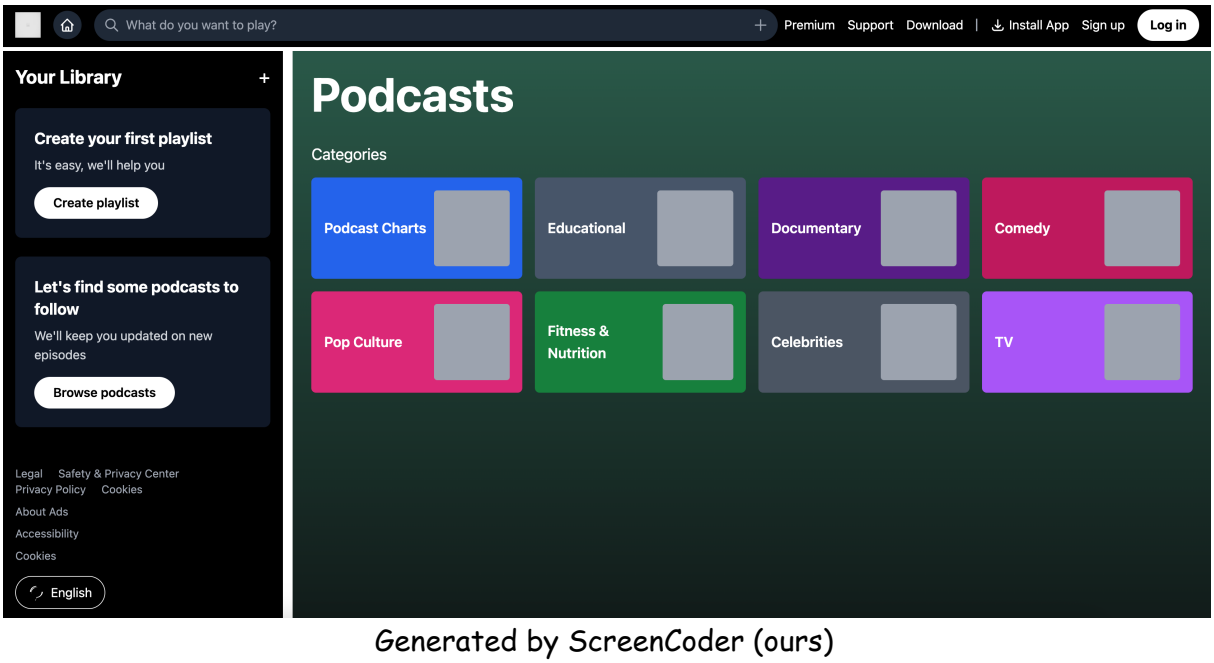


Figure 14: Qualitative comparison between our proposed method and various baselines.

Figure 15: **Failure Case Part 1: Input Design.** The input features a non-standard layout with a skewed container and rotated text.

Figure 16: **Failure Case Part 2: Qwen2.5-VL (Baseline).** The baseline suffers from *spatial hallucination*. It prioritizes pixel matching via fragile absolute positioning, leading to Z-index errors and overlapping text.

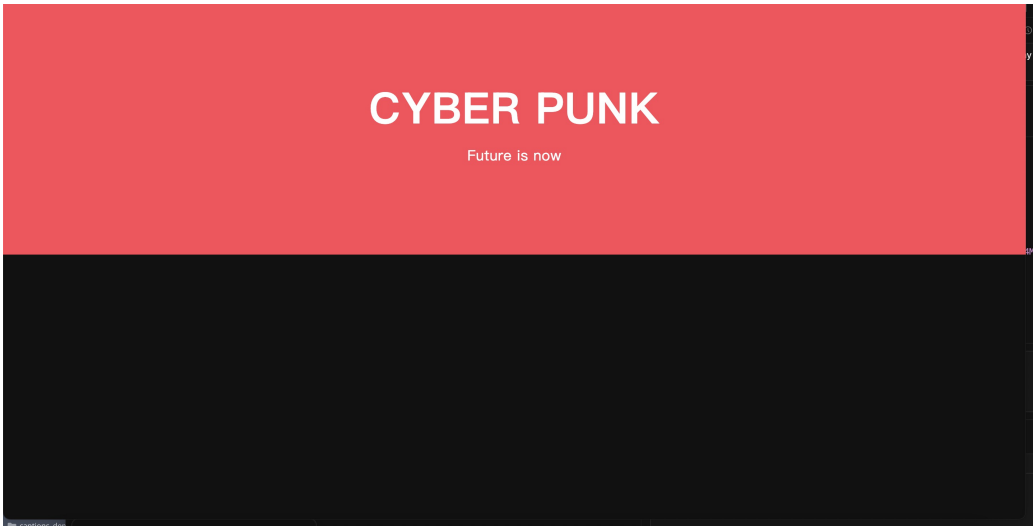


Figure 17: **Failure Case Part 3: ScreenCoder (Ours).** ScreenCoder simplifies the geometry to a standard rectangular grid (Graceful Degradation). While it misses the artistic skew, it maintains structural integrity, generating clean and valid code.