UIOrchestra: Generating High-Fidelity Code from UI Designs with a Multi-agent System

Chuhuai Yue[1], **Jiajun Chai**[1], **Yufei Zhang**[1], **Zixiang Ding**[1], **Xihao Liang**[1], **Peixin Wang**[1], **Shihai Chen**[1], **Wang Yixuan**[1], **Yanping Wang**[1], **Guojun Yin**[1], **Wei Lin**[1]*
[1]Meituan
yuechuhuai@meituan.com

Abstract

Recent advances in large language models (LLMs) have significantly improved automated code generation, enabling tools such as GitHub Copilot and CodeWhisperer to assist developers in a wide range of programming tasks. However, the translation of complex mobile UI designs into high-fidelity front-end code remains a challenging and underexplored area, especially as modern app interfaces become increasingly intricate. In this work, we propose UIOrchestra, a collaborative multi-agent system designed for the AppUI2Code task, which aims to reconstruct static single-page applications from design mockups. UIOrchestra integrates three specialized agents, layout description, code generation, and difference analysis agent that work collaboratively to address the limitations of single-model approaches. To facilitate robust evaluation, we introduce APPUI, the first benchmark dataset for AppUI2Code, constructed through a human-in-the-loop process to ensure data quality and coverage. Experimental results demonstrate that UIOrchestra outperforms existing methods in reconstructing complex app pages and highlight the necessity of multi-agent collaboration for this task. We hope our work will inspire further research on leveraging LLMs for front-end automation. The code and data will be released upon paper acceptance.

1 Introduction

Recently, large language models (LLMs) have significantly advanced the capabilities of various tasks. Leveraging novel learning paradigms and massive code corpus, state-of-the-art LLMs even surpass human performance in tasks such as code generation from natural language instructions (OpenAI, 2024b, 2025; Team, 2025; DeepSeek-AI, 2025a,b; Team et al., 2025). These developments have led to the emergence of code assistance tools like GitHub Copilot[1] and Amazon CodeWhisperer[2], which provide developers with professional-level code completions and snippets from simple descriptions or partial code, improving development efficiency. However, there remain important and in-demand tasks where LLMs are less effective.

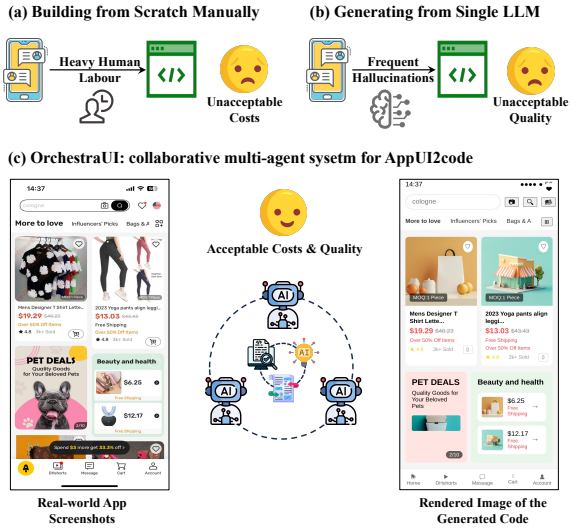

Figure 1: (a) Converting designs into static pages is a labor-intensive task. (b) One single LLM is prone to hallucinations when it comes to understanding UI designs. (c) Our proposed UIOrchestra helps developers achieve fast and efficient static page development.

The rise of mobile internet has led to a tremendous demand for app front-end development. Modern app interfaces typically feature complex layouts and numerous components. Front-end developers are required to accurately reproduce all elements specified in design mockups—including layout, hierarchy, and positioning—to construct static pages that faithfully reflect the original designs. While LLMs can assist with implementing interactive features, the initial process of static page construction remains tedious and time-consuming, resulting in

*Corresponding author.

[1]https://github.com/copilot
[2]https://aws.amazon.com/codewhisperer/

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 2769–2782
November 4-9, 2025

high development costs. Despite this demand, there has been limited research in this area. We refer to the challenging yet promising task of reconstructing static single-page apps from design mockups as **AppUI2Code**.

Some studies have attempted to generate simple web pages by converting website screenshots or randomly constructed prototypes into HTML (Zhou et al., 2024), a task commonly known as **UI-to-Code (UI2Code)** (Si et al., 2025). While these approaches show the potential of LLMs in automating labor-intensive tasks, the lack of relevant data during pre-training often results in hallucinations, especially in fine-grained details. This limitation is further amplified when the focus shifts from simple web pages to complex modern app interfaces, where the high density of UI components increases the challenge for a single model.

To address these challenges and enable efficient, high-quality static app page generation, we propose UIOrchestra (as shown in Figure 1), *a multi-agent system for AppUI2Code*, which consists of three agents: a layout description agent (A_{ld}), a code generation agent (A_{cg}), and a difference analysis agent (A_{da}). A_{ld} receives the UI design image and, guided by a few shot prompts, analyzes the layout in a row-by-row, divide-and-conquer manner, finally submitting a structural description of natural language. A_{cg} takes both the layout description and the design image as input, constructs the overall code framework based on the layout description, fills in style details according to the design image, and render the generated code. A_{da} compares the rendered page with the original design, identifies layout and style discrepancies, and iteratively feeds this information back to the code generation agent until the output is satisfactory.

To evaluate UIOrchestra, we examine existing benchmarks from prior work, which mainly focus on websites, finding them insufficient for assessing the reconstruction of complex app pages. To address this gap, we introduce **APPUI**, the first mobile-oriented benchmark for AppUI2Code. Unlike web front-end code, which can be easily collected at scale (e.g., Common Crawl, C4, WebUI), **mobile app front-end data is much harder to obtain**. Even large companies are limited to their own products, leaving other widely used apps inaccessible. To overcome this, we employed human experts within UIOrchestra, replacing A_{ld} and A_{da}in a human-in-the-loop process. Experts provided precise layout descriptions and difference analyses to A_{cg}, producing high-quality initial code samples. These samples were further refined by experienced engineers to ensure fidelity to the original designs. This process resulted in a validation dataset of 1,101 samples.

Experimental results reveal the limitations of single-model approaches and underscore the necessity of multi-agent collaboration for complex tasks like AppUI2Code. We hope future research will build on our work and, as model capabilities advance, further enhance the role of LLMs in front-end automation.

In summary, our contributions are as follows:

- We introduce UIOrchestra, a collaborative multi-agent system that automatically generates high-fidelity front-end code.
- We present APPUI, the first benchmark for AppUI2Code, by means of an expert intervention in UIOrchestra.
- Comprehensive evaluation on APPUI and other datasets demonstrates the advantages of UIOrchestra and clarifying the limitations of existing approaches.

2 Related Work

2.1 Code Generation from Natural Language

Recent years have seen rapid progress in code generation with both open- and closed-source LLMs advancing the state of the art. OpenAI's O series, leveraging reinforcement learning and chain-of-thought, has achieved strong results: O1 (OpenAI, 2024b) excels on Codeforces, and O3 (OpenAI, 2025) reaches 71.7% accuracy on SWE-bench Verified (Jimenez et al., 2024). Deepseek's open-source R1 (DeepSeek-AI, 2025a) matches O1's performance at lower training cost. Grok3 (Team, 2025) attains 79% on SWE-bench with significant resources. Other strong models include Moonshot AI's K1.5 (Team et al., 2025) and Qwen's QWQ (Team, 2024), both outperforming earlier models such as 4o (OpenAI, 2024a) and Claude (Anthropic, 2024).

2.2 UI2code

Before large language models (LLMs), early works like pix2code (Beltramelli, 2018) used CNNs and RNNs to generate code from UI screenshots, but were limited by model capacity. With the advent

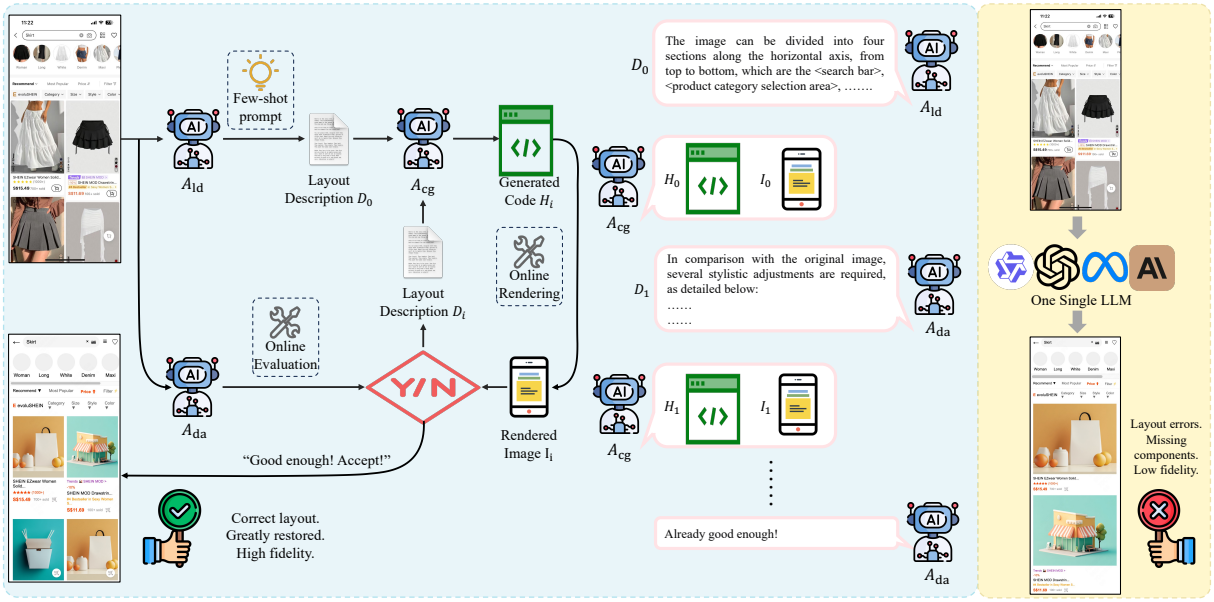

Figure 2: To reduce human involvement and lower the cost of data generation, we introduced a layout description agent and a difference analysis agent, forming UIOrchestra. The cooperation of multi-agents significantly enhances the quality of the generated data, surpassing that of a single model and enabling reliable automated data production.

of LLMs, UI2code (Guo et al., 2024) saw significant progress. Si et al. (Si et al., 2025) introduced a benchmark with 484 real-world webpages and tailored metrics for evaluating MLLMs on design-to-code tasks, inspiring much follow-up research. Laurençon et al. (Laurençon et al., 2024) released a dataset of 2 million HTML-screenshot pairs, while Gui et al. (Gui et al., 2024) curated a high-quality dataset from Common Crawl. Other advances include computer vision and compiler optimization (Zhou et al., 2024), "visual critic without rendering" (Soselia et al., 2024), and structure-aware attention with contrastive learning (Liang et al., 2024). However, most work targets websites and struggles with complex mobile pages. To address this, we propose UIOrchestra.

2.3 LLM-Based Multi-Agent System

While LLMs show strong capabilities, they still face challenges such as hallucinations. Recent research addresses this by leveraging multi-agent collaboration for collective intelligence (Tran et al., 2025). Studies have shown that combining different LLMs can enhance code generation (Barbarroxa et al., 2024). Frameworks like MetaGPT (Hong et al., 2024) and MapCoder (Islam et al., 2024) simulate human roles and programming stages to improve task performance. Other works (Shinn et al., 2023; Li et al., 2023; He et al., 2023; D'Arcy et al., 2024; Wang et al., 2024) further advance multi-agent systems by enhancing decision-making, collaboration, explanation, review, and recommendation through diverse agent interactions.

3 Building static pages via UIOrchestra

AppUI2code is a challenging and complex task that requires large models to possess strong visual capabilities, accurately understand the hierarchical relationships and layout structures implied by the various components on a page, and at the same time, not overlook the stylistic details of each element, including color, position, font, etc. Ultimately, all of this information must be faithfully converted into code. This entire process poses a significant challenge to the capabilities of LLMs. Therefore, given the limitations of current models, it is a reasonable approach to decompose the complex task into a series of sub-tasks, each handled by a dedicated agent, allowing for division of labor and collaborative problem-solving. Therefore, we propose UIOrchestra, which aims to generate static front-end code from UI design images in a step-by-step manner through multi-agent collaboration.

Specifically, as shown in Figure 2, we introduce a layout description agent, denoted as A_{ld}, and a difference analysis agent, denoted as A_{da}, complementing the code generation agent to form a specialized multi-agent system for AppUI2code. Collaboratively, these agents iteratively refine the

quality of the code. The input reference screenshot is referenced as I_0, while the front-end code produced by the code generation agent in the i^{th} iteration and its corresponding rendered image are referenced as H_i and I_i, respectively.

3.1 Layout Description Agent

A_{ld} takes I_0 as input and generates a detailed natural language description of its layout structure. As illustrated in Figure 2, A_{ld} analyzes the input page and produces an output description D_0. Guided by few-shot prompts, it partitions the UI page into regions through a top-down, left-to-right, line-by-line scanning approach. For example, a scan of the input in the figure sequentially identifies the status bar, search bar, category slider, and product detail area. Each sub-region is further decomposed in the same manner; for instance, the product detail area comprises a 2×2 grid of products, with each product containing vertically arranged components such as the product image, name, rating, and price. In this way, A_{ld} outputs the page's complete layout information D_0 in a tree structure, where the hierarchical level of each region or element reflects its position within the overall layout. By introducing A_{ld}, the crucial step of layout structure analysis is effectively isolated, which is essential for the AppUI2code task. This separation prevents A_{cg} from having to handle both layout analysis and code generation simultaneously, thereby reducing the risk of errors that may cause the final output code to deviate from the intended design. Furthermore, given the model's limited output length, dividing these tasks enables the model to generate more detailed layout descriptions, thus supporting a more accurate reconstruction of the original design.

3.2 Difference Analysis Agent.

A_{da} takes I_i and I_0 as input and is responsible for comparing their layout structures and visual details. Leveraging a similarity detection tool, A_{da} determines whether H_i requires revision. If so, it articulates the differences in natural language according to a predefined format to initiate a new iteration; otherwise, H_i is accepted as the final output. As illustrated in the figure, discrepancies may arise between I_i and I_0, such as: "In the original image, the product detail area is organized as a 2×2 grid. However, in the rendered image generated from the code, the product detail area contains only two products arranged vertically." By introducing a dedicated agent for defect detection, our approach not only facilitates the decomposition of complex tasks—compared to the self-refine prompt strategy in Design2Code (Si et al., 2025)—but also overcomes the limitations associated with relying on a single model.

3.3 Code Generation Agent.

A_{cg} takes I_0 and D_i as input and generates front-end code H_{i+1} according to the developer-specified programming language or framework, as the code generation rules may vary across different languages or frameworks. Subsequently, A_{cg} invokes an online rendering tool to obtain the corresponding rendered image I_i, which is then submitted to A_{da} for feedback. During generation, recognized elements such as images/icons will be replaced by predefined placeholder images, due to the inaccessibility of multimedia resources. In addition, A_{cg} is responsible for maintaining the complete context of the entire code generation process, ensuring that all previous outputs remain accessible throughout each iteration. Through this iterative code refinement process, the fidelity of the generated code is significantly improved compared to approaches that rely on a single model for code generation.

In summary, UIOrchestra leverages the collaborative strengths of multiple specialized agents to address the inherent challenges of the AppUI2code task. By decomposing the process into distinct, manageable sub-tasks and enabling iterative refinement, our approach not only enhances code fidelity but also improves the robustness and reliability of static page generation from UI design images. This multi-agent framework lays a solid foundation for further advancements in automated UI code generation.

4 Data Curation with Human-in-the-loop

As discussed in Section 1, existing benchmarks predominantly target web pages, which are often too simplistic to fully showcase the capabilities of UIOrchestra. Moreover, it is difficult to obtain front-end code for mobile applications, as such datasets cannot be easily gathered through web scraping. To address these challenges, we propose a human-in-the-loop approach that leverages UIOrchestra and expert intervention to generate front-end code that faithfully replicates original design images, resulting in APPUI—the first benchmark for AppUI2code—as shown in Figure 3. Notably, UIOrchestra is robust to different program-

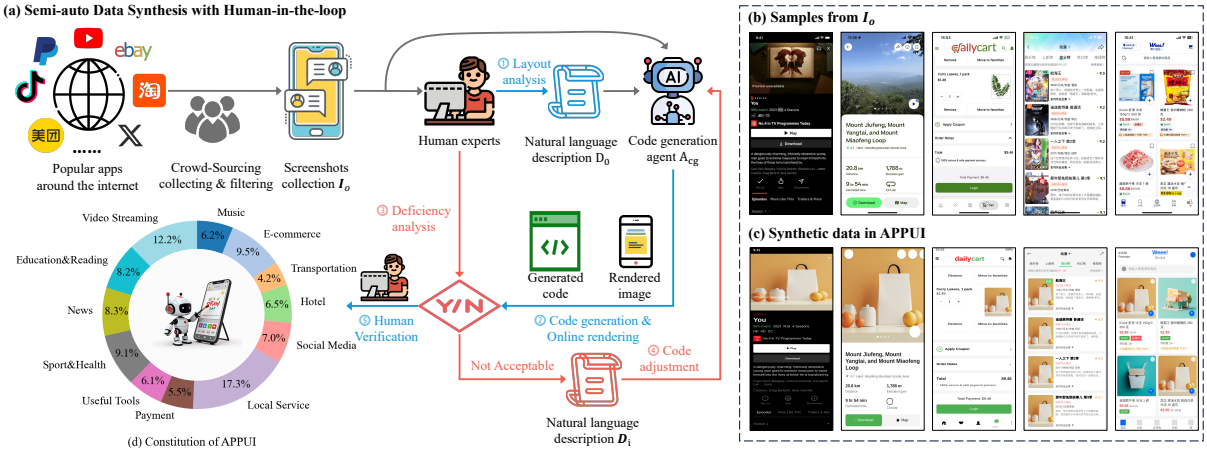

Figure 3: The construction of APPUI starts with collecting and categorizing screenshots from popular global applications. The code synthesis process involves expert-provided layout descriptions, code generation by A_{cg}, expert difference analysis, iterative correction, and final manual refinement. This results in 1,101 pairs of synthesized code and rendered images spanning 12 application categories.

ming languages and frameworks by simply modifying the prompt requirements. In this work, we opt to generate front-end code in the React framework, as it is widely used as DSL in practice. For example, React Native, which is extensively adopted, utilizes React during the development phase, and only maps to native components of the target system at the rendering stage on the app side through a dynamic framework.

4.1 Semi-auto High-Fidelity Data Synthesis

Our primary objective is to construct a dataset comprising screenshots of modern apps and their corresponding high-fidelity front-end codes, thereby filling the gap in the AppUI2code domain.

To obtain app screenshots, we engaged a crowdsourcing team to manually capture images from popular apps across commonly used categories, as automated methods often yield irrelevant pages and increase the burden of data filtering. Since there is no significant difference in app page styles between Android and iOS, all screenshots were collected from the iOS platform. Experts then screened the collected screenshots to remove those containing sensitive information, personal data, or pages unsuitable for image-to-code conversion (such as image-only pages). This careful selection resulted in our app screenshot set, denoted as $\mathbf{I_o} = \{I_o^1, I_o^2, ...I_o^K\}$, with examples shown in Figure 3(b). While screenshots are relatively easy to obtain, acquiring the corresponding front-end code remains a significant challenge.

As for corresponding codes, we employ a semi-automated human-in-the-loop approach, depicted in Figure 3(a). We apply A_{cg}, which driectly interacts iteratively with human experts rather than A_{ld} or A_{da}. For each sample, a human expert observes the UI screenshot and provides a completely correct layout description. Using this input, A_{cg} generates initial code. An online render tool displays the generated UI page in real-time, allowing the expert to compare it with the input sample and identify discrepancies. A_{cg} refines the code iteratively based on expert feedback until satisfactory fidelity is achieved. This process yields a preliminary set of code, $\mathbf{H}_{\text{ini}}$. On this basis, experts further refine $\mathbf{H}_{\text{ini}}$ through manual adjustments to ensure maximal consistency with the input pages. As the initial code quality is already high, this manual refinement process requires only minimal additional effort. The refined codes, $\mathbf{H}_{\text{ref}}$, are rendered to obtain screenshots, $\mathbf{I}_{\text{ref}}$. Thus, APPUI is defined as $\mathbf{APPUI} = \{S_1, \ldots, S_K\}$, where $S_i = \{I_{\text{ref}}^i, H_{\text{ref}}^i\}$, with examples shown in Fig3(c).

Figure 4: APPUI covers 12 categories of the most used app types in daily life, including the most popular apps in each category.

4.2 Data Statistics

Using the data synthesis method described previously, we filtered 2,000 screenshots from over 7,000 originals for synthesis, resulting in 1,101 pairs of front-end code and rendered images that make up the APPUI dataset. As shown in Figure 4, APPUI covers 12 common application categories, including e-commerce, education, travel, local services, music, news, payment, social media, sports, transportation, and video streaming. It includes 165 popular apps across these categories, with an average of 6.67 samples per app, focusing on the most frequently used pages. The average length is 2,199 tokens, measured by a BPE tokenizer. Figure 3(d) presents the distribution of categories, number of apps, average sample size, code length, and layout depth. Overall, APPUI offers broad and balanced coverage, capturing the diversity and complexity of modern app design styles without bias toward specific categories.

4.3 Objective Evaluation Metrics

Previous studies typically evaluate generated code as plain text, relying on metrics such as normalized edit distance. Design2Code, however, emphasizes visual fidelity by comparing the rendered images of generated and original code, introducing both high-level and low-level visual similarity metrics. In our work, we adopt both visual and textual evaluation metrics: visual metrics capture fidelity to the original design, while textual metrics reflect alignment with human coding conventions. Building on prior work, we employ objective metrics to assess both global and component-level visual and textual similarity.

For clarity, we denote the reference code and images as $\mathbf{H_t} = \{H_t^1, \ldots, H_t^K\}$ and $\mathbf{I_t} = \{I_t^1, \ldots, I_t^K\}$, and the generated code and images as $\mathbf{H_g} = \{H_g^1, \ldots, H_g^K\}$ and $\mathbf{I_g} = \{I_g^1, \ldots, I_g^K\}$. Take j-th output as an example.

To assess global visual similarity, we use the Structural Similarity Index (SSIM) and CLIP score. SSIM evaluates luminance, contrast, and structure, while the CLIP score measures similarity between image embeddings from the CLIP visual encoder, denoted as $\mathbf{CLIP}(I_t^j, I_g^j)$. For textual similarity, we compute BLEU scores between H_t^j and H_g^j.

For component-level evaluation, we establish a one-to-one correspondence between components in the generated and reference images. Let B_g^j and B_t^j denote the components of I_g^j and I_t^j, respectively, where each component includes textual content and positional coordinates. Unlike prior work that matches components solely by text, we incorporate spatial distance into the matching cost for optimal assignment, as detailed in the appendix.

A key goal of AppUI2code is to faithfully reproduce all elements from I_t^j in I_g^j. Following (Si et al., 2025), we use the Block-Match metric and propose the weighted mean CIOU score across all matched pairs as the primary component-level visual similarity metric:

$$\text{CIOU}_j = \sum_{(p,q)\in M^j} w_q \times \text{ciou}(\text{rect}_p, \text{rect}_q) \quad (1)$$

$$w_q = \frac{\text{Area}(\text{rect}_q)}{\sum_{i=1}^{m} \text{Area}(\text{rect}_i)} \quad (2)$$

where CIOU_j is the CIOU score for I_g^j, m is the number of matched elements, and rect_p is the bounding box of b_{tj}^q. The CIOU score directly reflects how well I_g^j restores the position and layout of each component.

For component-level textual similarity, since it is already considered during matching, we do not use it as a separate metric. Instead, we propose CSS-snippet matching, which extracts and compares CSS attributes (e.g., font size, color) from H_g^j and H_t^j for each matched pair. The metric is the ratio of correctly reproduced CSS fields to the total number of CSS fields across all pairs, capturing visual attributes beyond layout.

5 Evaluation

We conduct extensive benchmarking and human evaluation of various most recent powerful models and methods to explore the boundaries of their performance on our APPUI, including closed-source commercial models, open-source models, fine-tuned models, and our UIOrchestra.

5.1 Experiment Setup

The models under our evaluation include: proprietary models such as GPT-4o (OpenAI, 2024a), Claude-3.5 (Anthropic, 2024), Gemini-1.5 (DeepMind, 2024), Doubao-1.5 pro (DoubaoTeam, 2025), GLM-4v-plus (ZhipuAI, 2025); open-source models including Qwen2.5VL-7B (Bai et al., 2025), Llama 3.2-Vision 11B (MetaAI, 2024); fine-tuned models including Design2Code-18B (Si et al., 2025), WebSight VLM (Laurençon et al., 2024); as well as our UIOrchestra based on GPT-4o and Claude-3.5.

Methods	CLIP score	SSIM	BLEU	Block-match	CIOU	CSS snippet sim
GPT-4o	0.775	0.748	0.473	0.578	0.732	0.638
Claude-3.5	0.833	0.749	0.542	0.609	0.744	0.586
Gemini-1.5	0.604	0.623	0.426	0.474	0.589	0.541
Doubao-1.5pro	0.689	0.723	0.411	0.517	0.649	0.527
GLM-4	0.572	0.568	0.349	0.427	0.498	0.413
Qwen-2.5VL	0.643	0.719	0.425	0.479	0.593	0.518
Llama-3.2-V	0.619	0.681	0.386	0.472	0.531	0.437
Design2Code	0.584	0.617	0.394	0.439	0.487	0.424
WebSight	0.562	0.605	0.397	0.427	0.502	0.418
UIOrchestra(4o)	0.865	**0.776**	0.586	0.631	0.776	0.693
UIOrchestra(Claude)	**0.874**	0.769	**0.594**	**0.647**	**0.783**	**0.695**

Table 1: Objective evaluation results of various methods. The best results are in **bold**, second best are underlined.

5.2 Automatic Ojective Evaluation

As shown in Table 1, APPUI effectively distinguish the performance of different methods on the AppUI2Code task, offering a robust and objective assessment of model capabilities.

Commercial large models such as 4o and Claude achieve strong results across all metrics, benefiting from extensive multimodal pre-training. Notably, open-source models like Qwen2.5vl and Llama3.2v, despite their smaller sizes, perform comparably to earlier commercial models. In contrast, fine-tuned models based on web HTML data struggle to adapt to the density of elements in mobile app. Our proposed UIOrchestra addresses the limitations of single-model approaches by leveraging multiple specialized agents. Whether using 4o or Claude as the base, UIOrchestra consistently outperforms the original models, particularly at the component level, by capturing intricate details and minimizing distortion while adhering to coding standards.

Regarding evaluation metrics, CLIP score and SSIM assess semantic and structural similarity at the image level, with most advanced models performing similarly, though smaller models may introduce distortions. BLEU measures textual similarity, reflecting alignment with human coding preferences. Block-match and CIOU evaluate the accurate reproduction of component positions, though most models struggle with precise localization. CSS snippet similarity assesses the replication of visual details such as font and color, which significantly impact perception; even leading models like Claude show deficiencies here. UIOrchestra, through differential analysis, effectively captures and enhances layout, position, and color details, resulting in improved overall performance.

In addition to evaluating on our APPUI dataset, we also assessed UIOrchestra on the widely used web UI2code benchmark, DesignCode-hard (Si et al., 2025). We tested the performance of UIOrchestra with different base models, and the results are presented in Table 2.

It is worth noting that on the widely used web-based benchmark Design2Code (Si et al., 2025), most methods typically achieve high performance. In contrast, even state-of-the-art models exhibit significant room for improvement on our APPUI benchmark, demonstrating that APPUI presents a sufficiently challenging benchmark for the AppUI2code task.

	Block	Text	Position	Color	CLIP
GPT-4o					
Direct	56.6	89.8	78.6	81.9	87.1
Text-Augmented	67.7	95.2	77.5	81.5	87.5
Self-Revision	72.1	96.4	81.1	82.4	88.2
Claude-3.5					
Direct	61.7	91.1	83.0	84.4	89.5
Text-Augmented	75.1	97.6	83.4	84.9	89.0
Self-Revision	71.9	96.5	82.6	83.0	88.8
Gemini-1.5					
Direct	72.3	95.4	80.9	80.5	87.5
Text-Augmented	73.7	95.9	79.8	79.1	88.2
Self-Revision	71.2	96.6	80.9	78.4	87.9
UIOrchestra (4o)					
————	80.1	**98.6**	85.3	87.3	91.6
UIOrchestra (Claude)					
————	**82.5**	98.5	**86.0**	**88.9**	**93.1**

Table 2: The performance of various methods on Design2Code-hard (Si et al., 2025), with results of three commercial LLMs from its own paper. The best results are in **bold**.

5.3 Human Preference Evaluation

Methods	Text	Layout	Illustration	Detail	Total
GPT-4o	22	12	14	11	59
Claude-3.5	23	14	15	12	64
Gemini-1.5	20	11	13	12	56
Doubao1.5pro	22	13	12	11	58
GLM-4v	17	9	10	11	47
Qwen2.5VL	19	12	11	9	51
Llama 3.2V	17	10	12	10	49
Design2Code	15	12	9	7	43
WebSight	14	13	9	8	44
UIOrchestra	**24**	**17**	**19**	**16**	**76**

Table 3: Human preference evaluation results. The best results are in **bold**.

Given that the AppUI2Code task targets a broad user base, we designed an additional qualitative experiment involving human annotators. Specifically, we randomly selected 200 samples from APPUI, denoted as $\mathbf{I}_{\mathbf{t}}^{\mathbf{200}}$, and their corresponding output results from various methods. We invited four human annotators with design backgrounds, who each rated 1/4 of the outputs of all the methods, thus avoiding one person's favoritism towards a particular method. To avoid unexpected bias, they were unaware of the sources and were asked to score the outputs based on four predefined dimensions: text fidelity (correctness of text content recognition), layout fidelity (accuracy of layout structure), illustration fidelity (correct recognition and placement of images from the original screenshot), and detail fidelity (accuracy of colors, fonts, sizes, line breaks, etc.). Each dimension was scored out of 25 points and the final score of each dimension for each method is the mean of all the samples. The results of the experiment are shown in Table3. It can be observed that the qualitative results align well with the quantitative metric evaluations and our UIOrchestra achieved commendable performance in aligning with human preferences.

5.4 Fidelity Study for APPUI

To evaluate the gap between the synthesized data obtained in Section 4.1 and real app pages, as well as to assess the usability of APPUI samples for developers, we followed the methodology described in Section 5.3. Specifically, we invited human annotators to blindly evaluate the fidelity of $\mathbf{I}_{\mathbf{t}}^{\mathbf{200}}$ and the corresponding real-world screenshots (denoted as $\mathbf{I}_{\mathbf{o}}^{\mathbf{200}}$) according to the same criteria. The final scores were 25, 25, 24, and 23, with a total of 97 out of 100. These results indicate that, after multiple rounds of expert refinement, the samples in APPUI closely resemble authentic app screenshots.

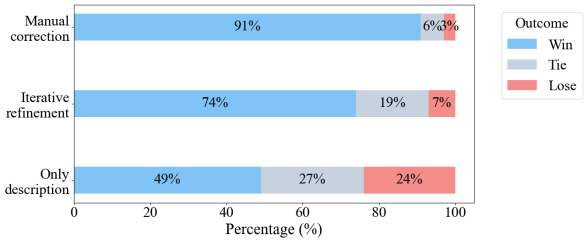

Figure 5: To validate the realism of APPUI samples, annotators blindly rated the similarity between rendered images and original screenshots. The baseline is set as single A_{cg}, with scores exceeding it by 10% marked as successes and those below by 10% as failures.

In addition, we conducted a comparative experiment to evaluate the data quality at each intermediate step mentioned in Section 4.1.

We first obtained the baseline result $\mathbf{H}_{\text{dir}}^{\mathbf{200}}$ by directly inputting $\mathbf{I}_{\text{o}}^{\mathbf{200}}$ into A_{cg}. The code generated using expert-provided layout descriptions is denoted as $\mathbf{H}_{\text{ld}}^{\mathbf{200}}$. The iteratively refined version by experts and A_{cg} is denoted as $\mathbf{H}_{\text{ref}}^{\mathbf{200}}$, and the final manually polished result as $\mathbf{H}_{\text{f}}^{\mathbf{200}}$. The same group of annotators then blindly evaluated the rendered images of $\mathbf{H}_{\text{dir}}^{\mathbf{200}}$, $\mathbf{H}_{\text{ld}}^{\mathbf{200}}$, $\mathbf{H}_{\text{ref}}^{\mathbf{200}}$, and $\mathbf{H}_{\text{f}}^{\mathbf{200}}$ using identical criteria. Scores exceeding the baseline by more than 10% were considered successful, while those falling below by more than 10% were considered failures. As shown in Figure 5, the quality of the generated code improves significantly as human expert involvement increases during the APPUI sample generation process. Ultimately, after meticulous manual review, an extremely high degree of fidelity to the input screenshots is achieved.

6 Conclusion

Our work highlights a missing capability in AI-assisted programming: generating static page code from design images. To address this challenging task, we propose UIOrchestra, a multi-agent collaborative framework that decomposes the problem into manageable steps. To rigorously evaluate UIOrchestra, we introduce APPUI, the first benchmark for the AppUI2code domain. Extensive experiments demonstrate the effectiveness of our approach. We hope this work will draw attention to this field, provide developers with more efficient tools, and promote further industry advancement.

Limitations

Due to the challenges in data acquisition discussed in the main text, it is difficult to obtain sufficient data samples to fine-tune open-source LLMs in this work. For the layout description subtask, fine-tuned LLMs may outperform general-purpose commercial models. In future work, we aim to leverage UIOrchestra to generate large-scale, high-quality datasets for fine-tuning and explore more solutions through post-training and related approaches.

References

Anthropic. 2024. Introducing claude 3.5 sonnet. https://www.anthropic.com/news/claude-3-5-son-net.

Shuai Bai, Keqin Chen, Xuejing Liu, and Jialin Wang et al. 2025. Qwen2.5-vl technical report.

Rafael Barbarroxa, Bruno Ribeiro, Luis Gomes, and Zita Vale. 2024. Benchmarking autogen with different large language models. In *2024 IEEE Conference on Artificial Intelligence (CAI)*, pages 263–264.

Tony Beltramelli. 2018. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '18, New York, NY, USA. Association for Computing Machinery.

Mike D'Arcy, Tom Hope, Larry Birnbaum, and Doug Downey. 2024. Marg: Multi-agent review generation for scientific papers.

Google DeepMind. 2024. Introducing gemini 2.0: our new ai model for the agentic era. https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/.

DeepSeek-AI. 2025a. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.

DeepSeek-AI. 2025b. Deepseek-v3 technical report.

DoubaoTeam. 2025. Doubao 1.5pro - doubao team. https://team.doubao.com/zh/special/doubao_1_5_pro. Accessed: 2025-03-08.

Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Yi Su, Shaoling Dong, Xing Zhou, and Wenbin Jiang. 2024. Vision2ui: A real-world dataset with layout for code generation from ui designs. *ArXiv*, abs/2404.06369.

Hongcheng Guo, Wei Zhang, Junhao Chen, Yaonan Gu, Jian Yang, Junjia Du, Binyuan Hui, Tianyu Liu, Jianxin Ma, Chang Zhou, and Zhoujun Li. 2024. Iw-bench: Evaluating large multimodal models for converting image-to-web.

Zhitao He, Pengfei Cao, Yubo Chen, Kang Liu, Ruopeng Li, Mengshu Sun, and Jun Zhao. 2023. LEGO: A multi-agent collaborative framework with role-playing and iterative feedback for causality explanation generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9142–9163, Singapore. Association for Computational Linguistics.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. Metagpt: Meta programming for a multi-agent collaborative framework.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues?

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

Huao Li, Yu Chong, Simon Stepputtis, Joseph Campbell, Dana Hughes, Charles Lewis, and Katia Sycara. 2023. Theory of mind for multi-agent collaboration via large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Shanchao Liang, Nan Jiang, Shangshu Qian, and Lin Tan. 2024. Waffle: Multi-modal model for automated front-end development.

MetaAI. 2024. Llama3.2: Revolutionizing edge ai and vision with open, customizable models.

OpenAI. 2024a. Gpt-4o system card. https://openai.com/index/gpt-4o-system-card/.

OpenAI. 2024b. Learning to reason with llms. https://openai.com/index/learning-to-reason-with-llms/. Accessed on 2025-02-25.

OpenAI. 2025. Openai o3-mini. https://openai.com/index/openai-o3-mini/. Accessed on 2025-02-25.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2025. Design2code: Benchmarking multimodal code generation for automated front-end engineering.

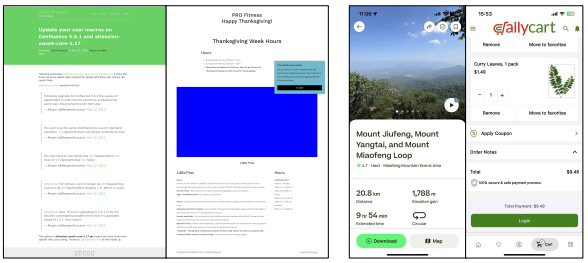

Figure 6: Website differs from mobile UI in component scale and layout structure

Davit Soselia, Khalid Saifullah, and Tianyi Zhou. 2024. Learning UI-to-code reverse generator using visual critic without rendering.

Grok 3 Team. 2025. Grok 3 beta — the age of reasoning agents. https://x.ai/blog/grok-3. Accessed on 2025-02-25.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, and Changjiu Jiang et al. 2025. Kimi k1.5: Scaling reinforcement learning with llms.

QwenLM Team. 2024. Qwq-max-preview: The next leap in deep reasoning and multi-domain mastery. https://qwenlm.github.io/blog/qwq-max-preview/.

Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D. Nguyen. 2025. Multi-agent collaboration mechanisms: A survey of llms.

Zhefan Wang, Yuanqing Yu, Wendi Zheng, Weizhi Ma, and Min Zhang. 2024. Macrec: A multi-agent collaboration framework for recommendation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR 2024, page 2760–2764. ACM.

ZhipuAI. 2025. Zhipu ai open platform. https://bigmodel.cn/dev/howuse/glm-4v. Accessed: 2025-03-08.

Ting Zhou, Yanjie Zhao, Xinyi Hou, Xiaoyu Sun, Kai Chen, and Haoyu Wang. 2024. Bridging design and development with automated declarative ui code generation.

A Is APPUI Necessary?

In the realm of UI2Code, prior work has predominantly focused on web-based applications, introducing datasets that range from large-scale fine-tuning datasets with millions of samples to high-quality benchmarks with a few hundred samples. However, these datasets are not directly applicable to the AppUI2Code task. Web pages generally exhibit a relatively uniform style, primarily centered around textual content with simple layouts, whereas mobile app interfaces are far more complex and sophisticated, featuring a greater diversity of styles. As illustrated in Figure 6, the sample on the left is from Design2code, while the one on the right is a screenshot of a real-world mobile app. The disparity arising from design philosophies effectively constitutes two distinct data distributions. Consequently, while existing web-based datasets may provide some foundational capabilities for the AppUI2Code task, they are not suitable for benchmarking purposes. Therefore, to advance the development of AppUI2Code, it is essential to propose a dedicated benchmark that reflects the unique characteristics of the task. Our work is motivated by this necessity.

B More samples from APPUI

To better demonstrate the diversity of data in the APPUI, we show more samples from it here, as shown in Figure 7.

C Components Hungarian Matching.

This algorithm is a two-phase UI component matching method. The first phase, the initial text similarity matching phase, involves constructing a similarity matrix by calculating the Levenshtein similarity of element names. The Hungarian algorithm is then used for preliminary optimal matching, and matches that exceed a certain threshold are selected. The second phase, the position optimization for elements with the same name phase, detects groups of elements with the same name in both documents. For groups with many-to-many matches, it constructs an Euclidean distance matrix based on spatial positions. A second Hungarian matching is applied to optimize spatial position matching, and the priority of optimized matching pairs is forcibly enhanced. This algorithm effectively addresses the issue of spatial position mismatches in scenarios with multiple elements having the same name, while maintaining the accuracy of text matching.

D Prompts for UIOrchestra

Syetem prompt of A_{ld}

I will provide you with a screenshot of a user interface, and I hope you can help analyze the layout structure of this image. I hope you can analyze it according to the following rules:

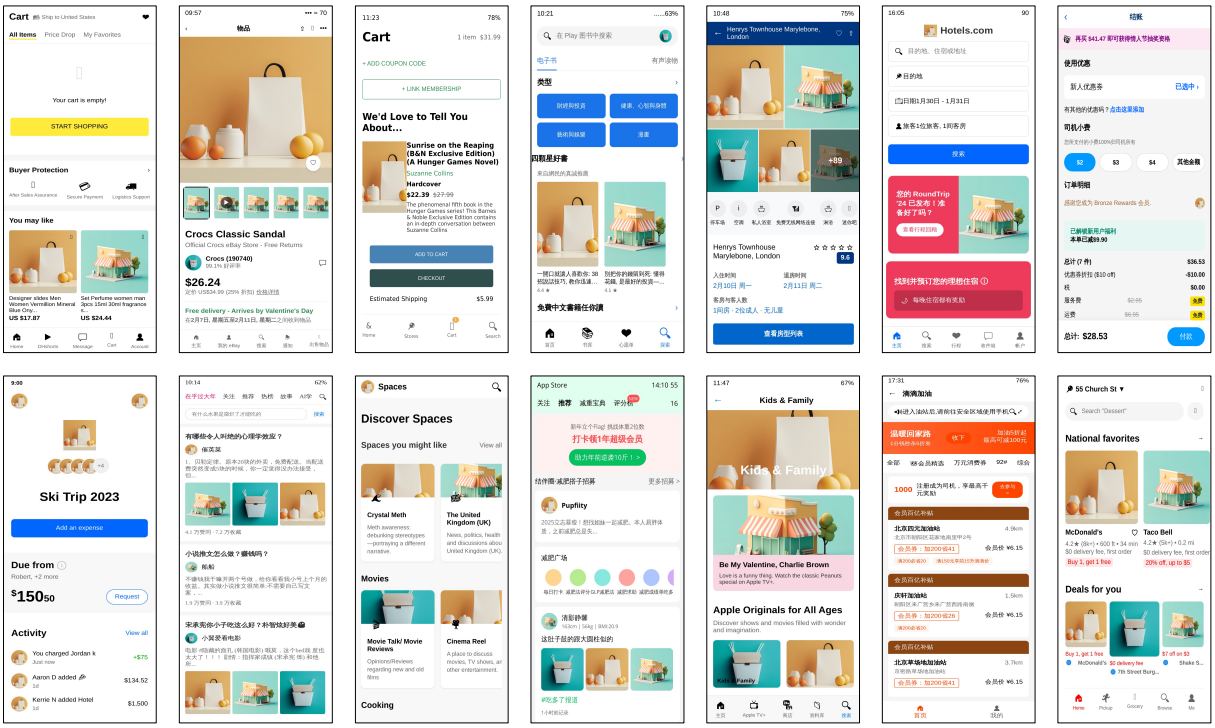

Figure 7: More samples from APPUI are displayed here.

Let's take it step by step:
1. Segmentation
Observe the image and determine whether the entire image can be divided into several parts by horizontal lines or vertical lines.
Segmentation instructions:

- When dividing, avoid having the dividing line pass through any UI component of the part to be divided, otherwise, bad things will happen.
- Choose one of the two segmentation methods.
- Summarize the divided parts with brief text.

2. Iterative Execution
For each of the divided parts, perform the first step for all parts that can be further divided until no fragments can be further divided.
3. Be sure to output according to the style in the example.
Example 1:
Input: url1
Output:
1. Segmentation
This image can be divided into two parts along the horizontal line: <Product Information>, <Store Information>from top to bottom.
2. Iterative Execution
For the <Product Information>area, it can be divided into two parts along the vertical line: <Product Image>, <Product Details>from left to right.

- For <Product Image>, no further division is needed, and its content is an image of a pizza.
- For <Product Details>, it can be divided into four parts along the horizontal line: <Product Name>, <Sales and Reviews>, <Honor Tags>, <Price Area>from top to bottom.
- .....

Example 2:
Input: url2
Output:
.....

Algorithm 1 Two-Phase Cross-Document Element Matching Algorithm

Require: GT component set $C^{gt} = \{c_1^{gt}, ..., c_m^{gt}\}$
Require: Gen component set $C^{gen} = \{c_1^{gen}, ..., c_n^{gen}\}$
Require: Similarity threshold τ
Ensure: Matching results $\mathcal{M} = \{(i, j, s)\}$

Initialize similarity matrix $S \in \mathbb{R}^{m \times n}$
for $i \leftarrow 1$ **to** m **do**
  for $j \leftarrow 1$ **to** n **do**
    $S[i, j] \leftarrow$ Similarity(c_i^{gt}.name, c_j^{gen}.name)
  end for
end for
Detect groups with the same name
$\mathcal{G} \leftarrow \{(name, I^{gt}, J^{gen}) \mid \exists i, j : c_i^{gt}.\text{name} = c_j^{gen}.\text{name} \neq \emptyset\}$
for each $(name, I^{gt}, J^{gen}) \in \mathcal{G}$ **do**
  if $|I^{gt}| > 1$ **and** $|J^{gen}| > 1$ **then**
    Construct distance matrix $D \in \mathbb{R}^{|I^{gt}| \times |J^{gen}|}$
    for $p \leftarrow 1$ **to** $|I^{gt}|$ **do**
      for $q \leftarrow 1$ **to** $|J^{gen}|$ **do**
        $d \leftarrow \|c^{gt}_{I^{gt}[p]}.\text{center} - c^{gen}_{J^{gen}[q]}.\text{center}\|_2$
        $D[p, q] \leftarrow d$
      end for
    end for
    (row_ind, col_ind) $\leftarrow$ HungarianAlgorithm(D)
    for $(p, q) \in$ (row_ind, col_ind) **do**
      $S[I^{gt}[p], J^{gen}[q]] \leftarrow 1.0$
    end for
  end if
end for
Cost matrix $C \leftarrow 1 - S$
(row_ind, col_ind) $\leftarrow$ HungarianAlgorithm(C)
$\mathcal{M} \leftarrow \{(i, j, S[i, j]) \mid (i, j) \in (\text{row\_ind}, \text{col\_ind}) \wedge S[i, j] \geq \tau\}$

System prompt of A_{da}

Role
You are a professional image analyst capable of precisely identifying detailed differences between two images and providing accurate adjustment suggestions.
Task
I will provide you with the original image and the preview image of AI generated front-end code.I need to know the differences between the preview image and the original image so that I can make further adjustments.
First, you need to accurately identify and analyze the visual information of the original and preview images, including but not limited to the position information of elements in the images, the relative layout of adjacent elements, detailed information of elements (text color, inner and outer margins, whether there are rounded corners, image aspect ratio, subcomponent layout, background color), etc.
Then you need to compare and analyze the visual detail differences of the preview image relative to the original image.
Finally, you need to provide me with accurate adjustment suggestions.
Requirements

- You must ensure that the content of the response is true and effective, and cannot mislead the user.
- You need to pay more attention to the fidelity of the preview image's details, and also consider the spacing and layout information between subviews.
- The adjustment suggestions provided should be as specific as possible, such as colors specified to color codes.
- Note: For differences in text size and font family, please ignore the related differences.
- For differences in rounded corners, you only need to focus on whether there are rounded corners, please ignore the size differences of the rounded corners.

- The images in the preview may use placeholders as fallback images; you do not need to focus on the specific content within the image components, only on the style and layout of the image components. If you find this difference, please ignore it directly and do not reflect this difference and adjustment suggestion in the response.

Response Format
You must respond strictly according to the following format:

By comparing with the original image, several styles need adjustments, with specific requirements as follows:

At xx point, there is a difference in xx, which requires xx adjustment.

At xx point, there is a difference in xx, which requires xx adjustment.

...

Syetem prompt of A_{cg}

Role
You are a professional AI UI2Code programming assistant with perfect visual capabilities, keen attention to detail, and extensive experience in React/CSS development, capable of accurately understanding UI designs and constructing layout structures into single-page applications.
Background
Users will provide UI design images that include various types of components, such as images, avatars, text boxes, etc. Users will inform you of the design dimensions, ensuring that the code accurately reproduces the layout. Users will provide you with layout structural descriptions, which you must refer to when generating code.
Task
Analyze the UI design image provided by the user and transform it into specified code. Specifically: You need to generate React code and corresponding CSS code.
Steps
Let's proceed step by step:

- Step One, Observe and analyze the design image to understand its content.

  Requirements:

  - Pay close attention to background color, text color, font size, font family, series, padding, corners, spacing, width, height, etc. Colors and dimensions must match accurately.
  - Focus closely on the relative positions of elements in the image to ensure that the generated code's original layout corresponds accurately to the input screenshot.
  - If there is text on the illustrations in the design, ignore it.

- Step Two, Carefully read and understand the input structural information, and generate code representing the hierarchical structure and layout relationships.

- Step Three, Combine the identified content to construct a single-page app using React framework.

  Requirements:

  - All components and containers must use relative layout.
  - Ensure the generated code appears identical to the input screenshot.
  - Use the exact text from the input image.
  - If the text in the image contains quotation marks, be sure to remove them.
  - For all images appearing in the design, use placeholder images (url:http://.....) and include detailed descriptions of the images in the alternative text.
  - More accurately identify the layout structure of images, including the overall layout and the layout of segmented sub-views.
  - Do not use comments to replace content code. Write complete code. Otherwise, something bad will happen.

- Repeat elements as needed to match the input image. For example, if there are 15 items, the code should have 15 items. You can use mapping and other iterative methods to render.
- For elements containing sub-elements, set the flex-direction value in CSS; otherwise, something bad will happen.
- CSS does not support group selectors like '.text1, .text2 ', it must be separated.
- Style elements should not be written in React code, but in CSS code.
- The length and width of containers should follow the user-input design dimensions.
- You must generate the React code according to the following template:

```
const App = () => {
    return <div className="
        container"></div>;
}
```

- Step Four, Review and optimize the code you have written.

  Requirements:

  - Ensure it can completely reproduce the style, especially layout positioning, otherwise something bad will happen.
  - Ensure the use of relative positioning and flex styles.
  - Check that the font size, color, and style in text boxes are consistent with the original image.
  - The code must adhere to the original image dimensions.
  - In the style code snippets, positions are specified using pixel units (px).