

## PROTOTYPE2CODE: END-TO-END FRONT-END CODE GENERATION FROM UI DESIGN PROTOTYPES

Shuhong Xiao<sup>1</sup>, Yunnong Chen<sup>1</sup>, Jiazhi Li<sup>1</sup>, Liuqing Chen<sup>1,2,\*</sup>, Lingyun Sun<sup>1,2</sup>, Tingting Zhou<sup>3</sup>

<sup>1</sup>College of Computer Science and Technology, Zhejiang University, Hangzhou, China

<sup>2</sup>International Design Institute of Zhejiang University, Hangzhou, China

<sup>3</sup>Alibaba Group, Hangzhou, China

### ABSTRACT

*UI-to-code technology has streamlined the front-end development process, reducing repetitive tasks for engineers. prior research mainly use design prototypes as inputs, with the effectiveness of the generated code heavily dependent on these prototypes' quality, leading to compromised robustness. Moreover, these approaches also exhibit shortcomings in code quality, including issues such as disorganized UI structures and the inability to support responsive layouts. To address these challenges, we introduce Prototype2Code, which achieves end-to-end front-end code generation with business demands. For Prototype2Code, we incorporate design linting into the workflow, addressing the detection of fragmented elements and perceptual groups, enhancing the robustness of the generated outcomes. By optimizing the hierarchical structure and intelligently recognizing UI element types, Prototype2Code generates code that is more readable and structurally clearer. To meet responsive design requirements, Prototype2Code primarily supports flexbox layout model, ensuring code compatibility across various device sizes. To validate the efficacy, we compare Prototype2Code with the commercial code generation platform CodeFun and Screenshot-to-code based on GPT-4 with vision. Employing structural similarity index measure (SSIM), peak signal-to-noise ratio (PSNR), and mean squared error (MSE) for visual similarity assessment, Prototype2Code's rendered UI effects align most closely with the design prototypes, exhibiting the minimal errors. We also conduct a user study with five experienced front-end engineers, inviting them to review and revise code generated by the three methods. As a result, Prototype2Code surpasses other methods in readability, usability, and maintainability, better meeting the business needs of industrial development.*

**Keywords:** UI Automation, Front-end Code Generation, Design Linting, Software Engineering

\*Corresponding author: chenlq@zju.edu.cn

### 1. INTRODUCTION

GUI-to-code generation has impacted the front-end industry, making development faster and cutting down on the repetitive work engineers encounter [1]. Leveraging deep learning advancements and extensive open-source datasets [2, 3], significant efforts are focused on developing techniques from visual representations, encompassing a fidelity from sketches [4], wireframes [5] to actual UI screenshots [6]. However, the stringent requirements of large-scale industrial development, including adherence to specific frameworks and the need for maintainable, accessible code [7], typically lead to a process that starts with the creation of design documents. To this end, platforms like Imgcook [8] and CodeFun [9] are utilized, serving as enterprise-level code generation platforms that cater to these stringent requirements.

The input design documents, also referred to as design prototypes, are crafted by UI designers utilizing professional software such as Sketch [10] and Figma [11]. Beyond visual representation, design prototypes incorporate a multimodal view hierarchy characterized by a tree-shaped layering structure, where each layer is endowed with attributes such as size, position, and type, thereby meeting the precise demands of development. However, the robustness of existing methods is often lacking, especially when designers do not consistently adhere to specifications, leading to lower quality designs. Such discrepancies in design quality, such as issues with fragmented elements and perceptual grouping, directly contribute to problems in the generated code. Existing solutions [12–16], while primarily focusing on design linting to influence the generation process at the input stage, exhibit notable limitations. On one hand, their effectiveness is constrained by the dependency on specific generation algorithms, limiting their compatibility and adaptability across different platforms. On the other hand, without addressing the complexities inherent in the generation process, they often result in visual discrepancies and structural errors in generated UI effects, requiring further manual adjustments for accuracy.

To address these limitations, we introduce Prototype2Code, an end-to-end solution that can automatically generate code from UI prototypes. Notably, we integrate the prototype linting (Fig. 1 a) as a crucial step in the generation process, with the goal of elevating design quality and, in turn, bolstering the robustness of the code produced. Following this, the insights gained during the linting process are leveraged to guide the construction of the UI layout tree (Fig. 1 b), ensuring that modifications made to the prototypes are accurately reflected in the final code structure. Furthermore, in constructing the UI layout tree, we take into account the industrial demand for responsive design, ensuring that the generated code is adaptable across devices of various sizes. By employing a graph neural network, we then determine the type of each UI element, such as text, image or list item (Fig. 1 c). Finally, in the generation stage, we utilize the layout tree and recognized element types to construct the HTML skeleton code and leverage large language model for generating CSS style code, thereby rendering the final UI effect (Fig. 1 d). To evaluate the effectiveness of our method, we conducted comparisons with results from generation platforms and other advanced approaches. Through quantitative visual similarity metrics, the UI effects rendered by our approach achieved the highest fidelity in reproducing the design intent. Our user study indicated that UIs generated by our method are more favored, and surpass other methods in terms of code readability, maintainability, and availability.

## 2. RELATED WORK

### 2.1 Code Generation Based on UI Prototypes

In large-scale frontend industrial development, collaboration between designers and engineers is essential. Designers produce UI prototypes, which frontend engineers then utilize as a basis for coding. Due to the prevalence of numerous similar elements and functionalities in frontend applications, engineers often spend significant time on writing repetitive code [1, 14]. To enhance efficiency, automatic code generation technologies for frontend have emerged. Compared to code generation based solely on visual representations, the path from design prototypes to code, enriched with multimodal information, better aligns with the requirements of industrial development [7, 13, 14].

It should be noted that, due to the need for design prototypes to be created by professional designers, obtaining them is much more challenging than acquiring visual representations. Most prototype to code methods originate from professional design platforms or large internet companies. For example, Figma to Code [17] and Builder.io [18] utilize engineering heuristic methods to parse prototype metadata for code generation. However, due to the lack of support for intelligent recognition of UI component types, their code fails to meet development specifications [7]. Yotako [19], powered by WordPress, primarily relies on templates to create personal blogs and sharing websites. Other platforms, such as Imgcook [8] and CodeFun [9], employ deep visual techniques to recognize component and layout structures, converting the outcomes into a generic domain-specific language (DSL). Subsequently, they apply distinct rules to generate frontend code with various frameworks. Despite the efficiency in meeting large-scale demands, the process is hindered by ongoing challenges such as UI element fragmentation and perceptual

grouping, necessitating further manual adjustments to the generated code. With large language models (LLMs) demonstrating exceptional performance in understanding requirements and generating code [20, 21], platforms like Anima [22] have integrated generative AI technology, enabling the creation of code through requirements expressed in natural language. However, the performance of these LLMs in layout inference still falls short of expectations, with the UI effects they generate not matching the design drafts accurately.

### 2.2 UI Fragmented Elements and Layers Grouping Problem

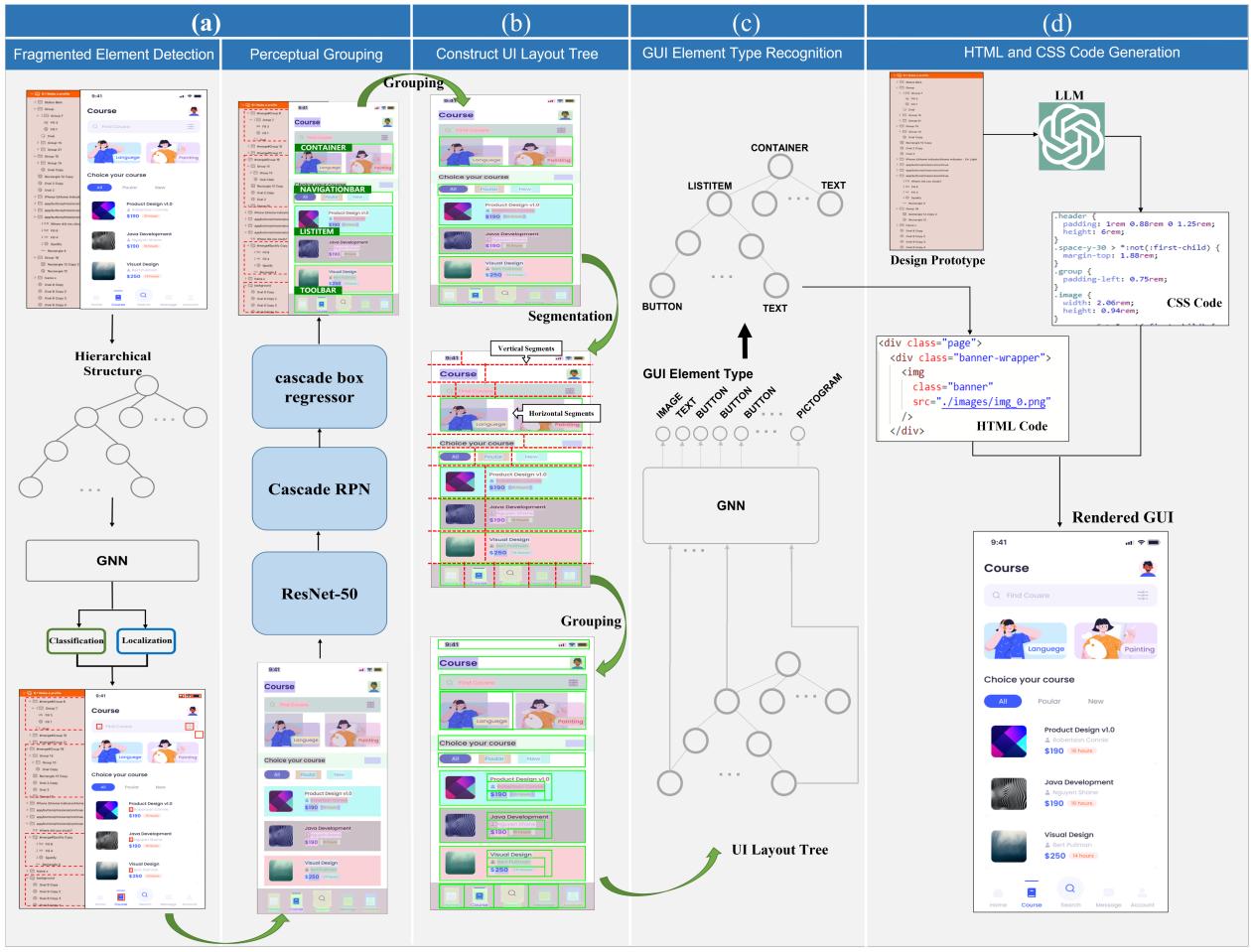
In design prototypes, designers construct a hierarchical view structure by stacking layers and form the overall UI effects. Take Sketch for instance, it supports layers of diverse attributes, including vector, text, bitmap, and shape. Also, in practice, not all layers convey tangible elements; some serve as organizational containers, integrating several layers with distinct properties to form cohesive groups. The issue of fragmented elements was first identified by Chen et al. [12]. A fragmented element group is typically composed of several basic vector or shape layers that together form a complete UI element, such as an icon. In the generated frontend code, these elements are not consolidated into a single icon image but exist as fragmented part instead, leading to redundant code structure and poor availability. They proposed a fragmented elements group detection method, using object detection techniques to locate potential groups and applying labels to the corresponding layers in the design prototypes to enforce consolidation. Li et al. [13] further clarified the concept of fragmented elements, defining them as layers that can not convey visual semantics independently. They primarily addressed the detection failures in previous visual methods caused by visual overlap among elements, such as the foreground and background occlusions in UI designs. Subsequent work, EGFE [14], extended their ideas by flattening UI elements in design prototype as a sequence and utilizing a Transformer to predict whether specific elements belong to fragmented elements.

The issue of layers grouping, as another focal point, revolves around discussing how to achieve UI grouping in a nested way, thereby enhancing the modularity of the generated frontend code and facilitating partial reuse. UIGDL [15], Screen Recognition [23] and UISCGD [16] addressed the identification of component-level groups, employing visual methods to locate groups containing the simplest UI functional units, such as a button comprising an icon and text. Inspired by the Gestalt principles, Xie et al. [24] further discussed the inference of section-level perceptual grouping, such as cards, lists, multi-tabs, and menus.

## 3. METHODOLOGY

### 3.1 Prototype Linting

Given a UI design prototype, its hierarchical structure unfolds in a tree-like manner, where each leaf layer node represents a specific UI element, such as text and image. Any non-leaf node encompasses multiple UI elements, forming specific perceptual areas and functional zones on the UI, such as lists and navigation bars. UI designs that do not strictly adhere to design standards



**FIGURE 1: Overview of Our Proposed Prototype to Code Approach**

may lead to deviations in this structure. In the issue of fragmented elements, a leaf node representing a fragmented element often fails to represent a UI element with complete semantics. For instance, a WiFi icon's three arcs-shape vectors might be represented by three separate leaf nodes. Regarding the perceptual grouping issue, some non-leaf nodes that indicate structure may be missing, causing the overall UI's nested structure to become less apparent. In the prototype linting section, we aim to achieve two objectives. First, we identify and mark fragmented elements that need to be merged; second, we perform perceptual grouping on the UI page to delineate regions. These information obtained is utilized in the second step to help construct the UI layout tree.

In this paper, we adopted the approach developed by Li et al. [13] to identify fragmented layers. Compared to other methods, Li et al.'s approach has stood out for its superior efficacy, particularly in surmounting the hurdles of inference in cases of UI element occlusion [12], while also remaining resilient against the pitfalls of erroneous data patterns caused by uneven distribution of layer types [14]. As illustrated in Fig. 1 a, the reconstruction of the hierarchical structure was initially undertaken: the original layering structure present in the design prototypes was abandoned in favor of reevaluating containment relationships, which were determined solely based on the position and size ( $x, y, w, h$ ) of each layer. Consequently, any parent node in the new layout tree was

designed to physically enclose all elements of its children in the UI.

Following this, we utilized a Graph Neural Network (GNN) for layer classification and bounding box regression. The classification branch was responsible for predicting whether a node belonged to a fragmented element, while the regression branch aimed at estimating the bounding box for the complete UI object associated with the current fragmented element. In the final step, groups of fragmented layers constituting a semantic UI element were marked. A merge attribute was added to their smallest parent node to indicate that these layers should be recognized as a single entity in subsequent processes. It is noted that all code and model weights from Li et al. were utilized, with all settings kept unchanged.

Following the identification of fragmented elements, we tackled the challenge of perceptual grouping of UI elements. This approach facilitated the creation of more structured and comprehensible code, thereby enhancing the performance and scalability of the generated front-end code in a more nuanced manner. Based on the CLAY dataset [3], we identified five types indicative of perceptual partitions: toolbars, navigation bars, cards, list items, and containers. The CLAY dataset, derived from the authoritative and widely used RICO dataset [2] in the UI field, has been optimized by removing noisy samples, thereby improving

the performance of our tasks. By filtering the CLAY dataset, we obtained a total of 50,524 pages and 318,631 objects. The dataset was divided into training, test, and validation sets in a 7:2:1 ratio.

Given that visual detection methods have been extensively validated for their effectiveness in UI scenarios, we adopted the SOTA detection method proposed by Chen et al. [12] to facilitate the inference of perceptual groups. They employed ResNet-50 [25] as the backbone to process UI visual representation, integrating Cascade RPN and a cascade box regressor [26] to incrementally refine the bounding boxes. To enhance detection performance, they introduced the boundary information of UI elements as additional knowledge to guide the inference process. We constructed the model based on the MMDetection [27] framework and trained it using the filtered CLAY dataset, with all parameter settings being identical to those reported by Chen et al. in UILM. As shown in Fig. 2, following this step, we have segmented the UI interface according to perceptual zones, providing guidance for the structure of frontend code, which is conducive to modular expression.

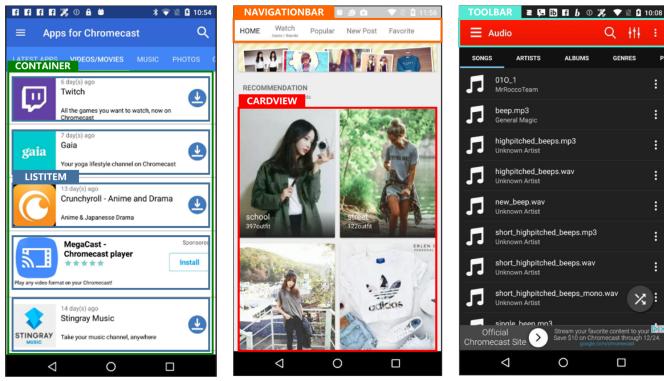


FIGURE 2: The Example of Perceptual Grouping Results

### 3.2 Constructing the UI Layout Tree

In this section, we explore the construction of the UI layout tree from design prototypes. Our approach is guided by two primary objectives: firstly, to address issues related to fragmented layers and the loss of perceptual groups; secondly, to ensure that the revised structure accommodates the demands of responsive layouts.

For all layers in the design prototypes, we start by reconstructing their containment relationships in the prototype based on the detection results of perceptual groups. We arrange all perceptual groups in ascending order of size and iterate them through all layers. This strategy is adopted to manage potential nesting issues within the UI, where larger perceptual groups may include multiple smaller ones. If the Intersection over Union (IoU) ratio between the current layer and a given perceptual group exceeds a predefined threshold, we consider the UI element represented by that layer to belong to the current group. Fragmented layers tagged with the `merge` attribute are treated as a single element, with their position and area calculated from the smallest top-left point to the largest bottom-right point coordinates. For UI elements that belong to the same perceptual group, we add a group label to their parent layer.



FIGURE 3: The Layout Structure Before and After Row/Column Re-segmentation

We then create the UI layout tree based on the optimized hierarchical structure. As illustrated in Algorithm 1, given a set of layers with hierarchical structure  $layer_i^N_{i=1}$ , we aim to output a tree structure  $T$  that consists of a set of nodes  $V$  and a node relation matrix  $M \in \mathbb{R}^{N \times N}$ ,  $M[i, j] = 1$  indicates that node  $i$  is the parent node of node  $j$ . Our entire algorithm is divided into two phases: layout tree initialization and row/column re-segmentation for flexbox layouts. For the layout tree initialization, we extract all leaf layers representing UI elements as the set  $E$ . For those belonging to fragmented layers or part of perceptual groups, we take out the element represented by their parent node as a whole representation. For all the UI elements obtained in  $E$ , we sort them in descending order by their size and iterate through them, adding the current element to the tree under the node with which it has a containment relationship and the smallest size. Following this, we deal with the overlapping issue among UI elements, which affects the generation of layout-related CSS code. For each non-leaf node in  $T$ , we check its subordinate nodes for overlaps and generate new parent nodes for them with overlapping, subsequently attaching it to their original parent node. We also add an attribute `need_absolute=True` to the newly generated parent nodes, indicating that the elements under these nodes require absolute positioning in their layout.

In order to achieve a responsive design, we utilize flexbox layouts for UI elements. This requires that UI elements under a parent node have a uniquely determined arrangement and their order can be defined. Fig. 3 a demonstrates a structural effect up to this point. Within the green-boxed parent node, elements in red boxes are inconsistently arranged, some vertically and others horizontally. Therefore, in the second phase of Algorithm 1, we conduct row/column re-segmentation to further refine the groups, ensuring that elements within a group are explicitly ordered either by rows (left to right) or by columns (top to bottom). Generally, this involves understanding the relative positions of elements on a UI page, which has been widely discussed in the field of UI accessibility. In this case, we enhanced the segmentation algorithm from Screen Recognition [23], which is based on the reading order from top to bottom and left to right, to determine the order of elements on a page as we required. In summary, the process

---

**Algorithm 1** UI Layout Tree Construction Algorithm

---

**Input:** the prototype structure  $\{layer_i\}_{i=1}^N$

**Output:** the UI layout tree  $T = \{V, M\}$

Phase 1: Layout tree initialization

```

1:  $E = T.\text{LeafLayers}().\text{map}(L \rightarrow$ 
    $L.\text{isFrag} \parallel L.\text{isInGroup} ? L.\text{parent}() : L)$ 
2: for  $e$  in  $E.\text{descending}()$  do
3:    $V = V \cup \{e\}$ 
4:    $M[e.\text{findparent}(T), e.\text{index}()] = 1$ 
5: end for
6: for  $e$  in  $T.\text{NonLeafLayers}()$  do
7:   for  $O = e.\text{Overlap}().\text{Empty}() ? e.\text{Overlap}() : \{\}$  do
8:     for each element set  $S$  in  $O$  do
9:        $V = V \cup \{\text{NewNode}(\text{need\_absolute} = \text{True})\}$ 
10:       $M[V.\text{len}(), i] = 1$  for  $i$  in  $S$ 
11:    end for
12:   end for
13: end for
Phase 2: Row/column re-segmentation
14: repeat
15:   Create vertical segments based on  $Y$  coordinates
16:   Create horizontal segments based on  $X$  coordinates
17: until No more segments can be added
18: for each segment  $S$  in vertical segments do
19:    $V = V \cup \{\text{NewNode}(\text{flex\_direction} = \text{row})\}$ 
20:    $M[V.\text{len}(), i] = 1$  for  $i$  in  $S$ 
21: end for
22: for each segment  $S$  in horizontal segments do
23:    $V = V \cup \{\text{NewNode}(\text{flex\_direction} = \text{column})\}$ 
24:    $M[V.\text{len}(), i] = 1$  for  $i$  in  $S$ 
25: end for
26: return the UI layout tree  $T = \{V, M\}$ 

```

---

is divided into three steps: (i) We create vertical segments by identifying  $Y$  coordinates at which a horizontal line can be drawn across the page without intersecting any elements. (ii) Similarly, within these vertical segments, we form horizontal segments using  $X$  coordinates, following the same non-intersecting principle. (iii) This first two steps are recursively applied to all horizontal segments until no more segments can be added. Then, based on the segmentation results, we determine the horizontal layout among UI elements. If multiple UI elements are within the same horizontal segment, we group them together under a new parent node, setting its flex layout property to `flex-direction=row`, as illustrated by the elements “\$190” and “16 hours”. Subsequently, we examine the vertical layout. In cases where multiple UI elements are in the same vertical segment, we group them and set the flex layout property to `flex-direction=column`. The final effect (Fig. 3 b) allows us to ascertain the arrangement and order of any element relative to its parent node.

### 3.3 UI Element Recognition

So far, we have laid the groundwork for generating the HTML code. In this section, we explore the process of assigning specific element types to each UI element within the UI layout tree. These types, such as “icon”, “status-bar” and “button”, are utilized as

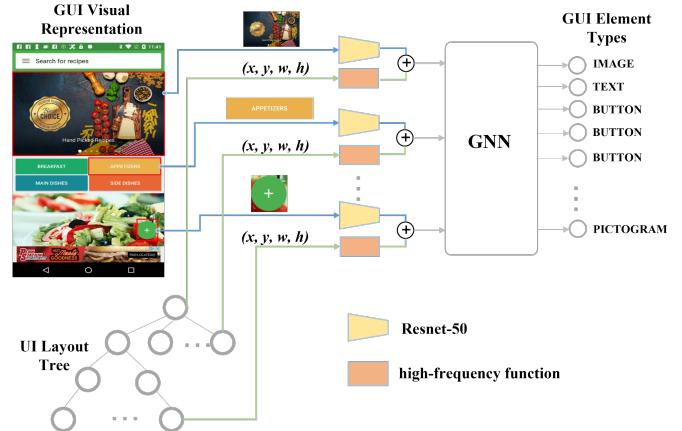


FIGURE 4: Our Approach to Predict UI Element Types

class names for each `<div>` element. By implementing this approach, we enrich the semantics of the HTML code, thereby making it more readable and promoting the reuse of specific components [7].

Figure 4 presents our approach for the recognition of UI element. Reflecting on the insights from prior research [14, 28, 29], which highlighted the value of multimodal information in enhancing UI element identification, our method integrates both the visual representation and spatial dimensions  $(x, y, w, h)$  of each element. For each node in our UI layout tree, we extract its visual representation, scale it to a size of 64x64 pixels, and employ ResNet-50 to obtain image feature  $I$ . Regarding the spatial dimensions information, we encode it into a high-dimensional vector  $S \in \mathbb{R}^L$  using the following high-frequency function:

$$\gamma(z) = (\sin(2^0 \pi z), \cos(2^0 \pi z), \dots, \sin(2^{L-1} \pi z), \cos(2^{L-1} \pi z)). \quad (1)$$

Here,  $\gamma(z)$  maps the position and size information of each UI element one-to-one into a higher-dimensional feature space, facilitating better integration with other features during subsequent forward propagation. We embed both the image feature  $I$  and the spatial feature  $S$  into 128-dimensional vectors, and apply element-wise addition to combine them. This sum serves as the initial state information for each node within the graph neural network(GNN).

To address the limitations of GNN such as oversmoothing and oversquashing, we draw inspiration from prior work [30, 31] and incorporate a multi-head self-attention mechanism into our model as shown in Equation 2. Specifically,  $X_T^{L+1}$ , representing the state of the nodes at the  $(L+1)^{th}$  layer of the network, is composed of two parts:  $X_M^{L+1}$  originated from the graph network itself and  $X_T^{L+1}$  derived from the multi-head attention mechanism.

$$\begin{aligned}
 X^{L+1} &= \text{MLP}^L(X_T^{L+1} + X_M^{L+1}), \\
 X_T^{L+1} &= \text{Self-attn}^L(X^L), \\
 X_M^{L+1} &= \text{MPNN}^L(X^L), \\
 X_M^{L+1} &= \text{LayerNorm}(\text{Dropout}(X_M^{L+1}) + X^L), \\
 X_T^{L+1} &= \text{LayerNorm}(\text{Dropout}(X_T^{L+1}) + X^L).
 \end{aligned} \quad (2)$$

In this context, MLP refers to a multi-layer perceptron, a neural network module where each layer is fully connected to the subsequent one. The self-attention module, crucial in Transformer architectures, assigns weights to different parts of the input, enhancing focus on relevant features. The MPNN (Message Passing Neural Network), facilitates effective graph learning by enabling the exchange of messages among nodes via their connecting edges [32]. LayerNorm stands for layer normalization, and dropout is employed to prevent overfitting in the learning process.

After  $K$  rounds of GNN state updates, we employ a two-layer MLP to accomplish the recognition of element types. We trained the model on the CLAY dataset [3], comprising 59,555 UI pages with 1,368,383 objects across 25 types. The dataset was split into a 7:2:1 ratio for training, testing and validation. The initial learning rate was set at 1e-4, with a decay rate of 0.99. Due to the imbalance in UI element types in dataset, we adjusted the weights of the cross-entropy loss according to the frequency of types in the training set, setting the weight to be the inverse of the frequency.

### 3.4 HTML and CSS Code Generation

Finally, in this section, we introduce the method for generating UI front-end code, which is primarily divided into two parts: the HTML skeleton and the CSS style code. With the development of multimodal large language models(LLMs) in the field of code generation, recent studies have explored their performance in front-end code generation. Among these, the open-source tool Screenshot-to-Code [33] has emerged as one of the most popular and representative one. Despite its capabilities, Screenshot-to-code still lacks the ability to parse input GUIs accurately, limiting it to generating rendered code that only stylistically matches the design input. Learning from these insights, we decided to separate the tasks of layout generation (HTML) and style generation (CSS) (Fig. 1 d). This approach leverages the strengths of LLMs in text generation for CSS code production, while also maintaining the UI layout tree’s strong guidance for the HTML code structure.

We first demonstrate how to derive HTML code from the UI layout tree. For any non-leaf node, we encapsulate it with a `<div>` tag and traverse the subtree rooted at it, using the nesting of `<div>` tags to depict the entire skeleton layout. For leaf node elements representing images and text, we apply `<img>` and `<span>` tags, respectively. Utilizing the results of UI element category recognition, we assign a classname to each element to serve as a CSS selector.

CSS styles dictate the rendering effects of elements on a page. There are two types of CSS styles: layout-related properties determine the rendering position of elements on the page; visual effect-related properties, on the other hand, ensure the UI’s aesthetic appeal. Previous research [9, 33] have predominantly supported absolute size layouts, leading to pages that cannot respond to device sizes, which is not conducive to further development and maintenance. With this consideration, we adopt a flex layout model that can respond to device sizes. For each element wrapped in a `<div>`, we determine the layout method based on the attributes set in Section 3.2. For nodes with the `need_absolute` attribute, we set their CSS position prop-

erty to `relative`, and their children’s `position` property to `absolute`. Here, child nodes are absolutely positioned based on the top-left corner coordinates of the parent node, setting their `top` and `left` properties. If a child node fits the characteristics of row or column arrangement, we set the parent node’s `position` property to `flex`, and arrange it in rows or columns according to the `flex-direction` property determined in Section 3.2. The `margin-top` and `margin-left` properties of each child node are determined by their distance from the parent node or adjacent nodes. The `width` and `height` properties can be directly obtained based on parameters in design prototypes.

Finally, we describe how to utilize LLMs such as GPT-4 [34] for generating the remaining visual effect-related CSS properties. Our prompt comprises four parts: `<Role-playing, User Input, Field Explanation, Output Requirement>`. In the role-playing part, we prompt the LLM as a front-end developer with years of experience and advanced technical skills. For user input, based on the Figma API user documentation [35], we filter out style information unrelated to visual effects, organizing the remaining information in a dictionary key-value pair format. In the field explanation part, we review the Figma API documentation again, storing interpretative texts for all relevant style properties as a JSON file, enabling the LLM to understand the style in the design prototypes. In the output requirement, we instruct the LLM to return a parsable JSON, where each field corresponds to a related CSS property and its value.

## 4. EXPERIMENTS AND RESULTS

### 4.1 Perceptual Grouping Performance

First, we assess the performance of our perceptual group detector. The advantage of the cascading network lies in its ability to refine bounding boxes through multi-stage regression, reducing the noise from low-quality candidate boxes while also preventing overfitting. As shown in Table 1, we report the object detection performance using the standard COCO metrics [36]. Considering the significant size variations among UI elements, we also provide separate Average Precision (AP) scores for large ( $> 96^2$  pixels), medium ( $32^2 - 96^2$  pixels), and small ( $< 32^2$  pixels) targets.

For comparison, we include three major types of object detection models as baselines. The Faster-RCNN [37] and its optimized variant, GA-Faster-RCNN [38], represent the two-stage anchor-based bounding box refinement approach (Type 1). CenterNet [39] and YOLOF [40], which are anchor-free and operate through end-to-end bounding box regression (Type 2), are similar in model size to our detector and have demonstrated commendable performance on the COCO dataset. Lastly, Deformable-DETR [41], which is built upon the Transformer architecture (Type 3), serves as a quintessential example of this type in the object detection framework.

As a result, our detector, categorized within Type 1, surpassing both Faster-RCNN and GA-Faster-RCNN with an increase of 5.7% and 2.3% in mean average precision (mAP), and approximately a 2% improvement in precision for detecting small targets. Among the second type of models, CenterNet performs less effectively, trailing our detector by 7.3% in mAP. The YOLOF model shows better results but falls short in precision for small

TABLE 1: Results of Perceptual Group Detection (IoU 0.50:0.95)

Method	<i>mAP</i>	<i>AP<sub>s</sub></i>	<i>AP<sub>m</sub></i>	<i>AP<sub>l</sub></i>
Faster-RCNN	0.583	0.197	0.287	0.597
GA-Faster-RCNN	0.617	0.200	0.306	0.629
CenterNet	0.567	0.179	0.297	0.576
YOLOF	0.617	0.107	0.266	0.633
Deformable-DETR	0.640	0.126	0.279	<b>0.655</b>
Ours	<b>0.640</b>	<b>0.219</b>	<b>0.369</b>	0.651

and medium elements. Similarly, Deformable-DETR, as the third type of model, show less precision in grouping small elements, with an 11.2% lower performance compared to our detector. Overall, its *mAP* is roughly on par with our model, and it slightly surpasses our model by 0.4% in recognizing large targets. However, Deformable-DETR requires more VRAM and longer training times in terms of training resources.

## 4.2 Element Type Recognition Performance

In this section, we report on the performance of UI element type recognition. Identifying the types of UI elements is a crucial step in the front-end code generation process, as it enhances the semantic richness of the code and improves its readability. As illustrated in Table 2, we evaluate the model using three metrics:

$$\begin{aligned} \text{precision} &= \frac{TP}{TP + FP}, \\ \text{recall} &= \frac{TP}{TP + FN}, \\ F1 \text{ score} &= 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}. \end{aligned} \quad (3)$$

Given the CLAY dataset's composition of 25 UI element types and its imbalanced sample distribution, we present not only the default macro average but also the results of the weighted average to provide a comprehensive performance overview.

In the task of recognizing UI element types, certain elements, like text elements and text buttons, exhibit high visual similarity, potentially causing neural networks to make incorrect type classifications. To address this issue, we employ a Graph Neural Network (GNN) to achieve a comprehensive feature representation for each element. Utilizing the message-passing capabilities of GNN allows us to capture the contextual and semantic relationships between elements within the GUI layout. This method significantly improves the uniqueness of the representation vectors, leading to enhanced accuracy in type classification. As a validation, we compare our method with a baseline model excluding GNN (line 1 in Table 2). The inclusion of GNN has led to noticeable improvements: a rise in the weighted average metrics—precision, recall, and F1 score by 4.3%, 4.4%, and 4.8% respectively. While a slight decrement in macro average precision is observed, significant enhancements have been observed in recall and F1 score, each by around 5%.

Another issue that has garnered attention lies in the expression of elements' visual representation. In our method, we uniformly scale all UI elements to a size of 64x64 pixels which is a common practice applied. However, UI pages often contain

TABLE 2: Results of Element Type Recognition

Method	Weighted Average			Macro Average		
	precision	recall	F1 score	precision	recall	F1 score
ResNet	0.759	0.761	0.754	<b>0.710</b>	0.591	0.628
ResNet+GNN (ours)	<b>0.802</b>	<b>0.805</b>	<b>0.802</b>	0.708	<b>0.649</b>	<b>0.673</b>
ResNet+RoI+GNN	0.737	0.739	0.727	0.641	0.526	0.555
ResNet+FRN+GNN	0.733	0.737	0.723	0.669	0.525	0.568

elements that are either too wide or too narrow. For instance, a toolbar may have an aspect ratio exceeding 6. In such cases, uniform scaling can distort the elements, leading to a loss of semantic information and poor representation. To avoid information loss due to image scaling of components, previous methods have employed a strategy combining RoIPooling and RoIAlign [13, 42] to obtain feature maps that match the original dimensions of the elements. To validate the performance of this strategy, we introduce two additional baselines. The RoI method (line 3 in Table 2) extracts the feature map of the entire UI using ResNet-50, and then employs the RoIAlign algorithm to obtain representations for each single element region. For the FPNRoI method (line 4 in Table 2), it further enhances the FPNRoI method by utilizing a Feature Pyramid Network (FPN) to integrate high-dimensional and low-dimensional feature expressions of the UI page. From the results, the approach of uniformly scaling to extract elements features (our method) proves to be more suitable for UI element recognition. Compared to the suboptimal RoI method, it achieves improvements in the weighted average precision, recall, and F1 score by 6.5%, 6.6%, and 7.5%. A possible reason may stem from the data-centric nature of the model: since ResNet-50 is pretrained on the ImageNet dataset, which does not contain UI images, the features it extracts may not effectively match those found in UI elements.

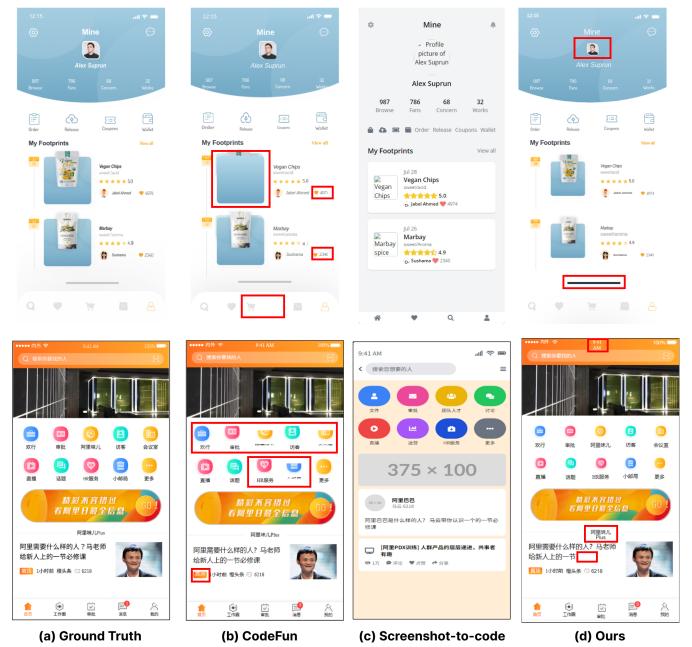


FIGURE 5: Comparison of Rendered Effects

### 4.3 Visual Quality Assessment

Assessing the alignment of rendered GUI visual effects with original design intentions is crucial for evaluating front-end code generation. In this section, as shown in Table 3, we introduce three metrics based on visual similarity for the consistency evaluation. Among them, structural similarity index (SSIM) evaluates from the perspectives of luminance, contrast, and structure, yielding results that closely mirror the intuitive perceptions of human; peak signal-to-noise ratio (PSNR) and mean squared error (MSE) measure absolute errors, assessing image similarity based on the differences in corresponding pixel values. Higher SSIM and PSNR values indicate that the image quality is closer to the true value, while for MSE, a lower value signifies better consistency.

We select two baseline models for our comparison: CodeFun [9] is a commercial code generation platform that supports large-scale industrial development while Screenshot-to-code [33] is currently the most popular open-source generation tool, implemented based on GPT-4v. It is important to note that Screenshot-to-code only supports code generation from UI screenshots and, compared to our method and CodeFun, lacks the multimodal information contained in design prototypes, making the comparison somewhat unfair. However, considering that the use of multimodal LLMs to assist in front-end code generation may become a trend in the future, we include it for comparison and discuss the existing shortcomings.

Regarding the dataset, we select 10 design prototypes from the fragmented elements dataset by Chen et al. [12], which contains a large number of design errors, representing outputs from relatively low-skilled designers. Additionally, we acquired another 20 free designs from the Figma community, each averaging over 200 likes, representing higher-quality designs. Statistically, we conducted validations on 30 design prototypes, encompassing a total of over 4000 layers.

TABLE 3: Results of Visual Quality Assessment

	SSIM ↑	PSNR ↑	MSE(1e-2)↓
<b>Ours</b>	<b>0.91 ± 0.07</b>	<b>21.15 ± 2.01</b>	<b>0.97 ± 0.42</b>
<b>CodeFun</b>	0.64 ± 0.06	17.42 ± 2.26	3.57 ± 1.45
<b>Screenshot-to-code</b>	0.58 ± 0.12	0.58 ± 0.12	10.46 ± 2.35

Quantitatively, the UI effects rendered by our method are significantly superior to those of CodeFun and Screenshot-to-code, with improvements of 0.27 and 0.33 in SSIM; 3.73 and 9.14 in PSNR; and a reduction of 2.6 and 9.49 in MSE, respectively. Qualitatively, we demonstrate the differences in two rendered effects as Fig. 5. We have marked the evident errors with red boxes. Comparing CodeFun with our method, the former exhibits more errors, with its image and text elements showing incomplete displays. In contrast, the discrepancies in our method primarily stem from differences in the applied CSS visual properties, which can be rectified by adjusting the target object. Unlike other methods, Screenshot-to-code does not achieve a faithful replication of designs; its code-rendered images bear only a stylistic resemblance to the original designs. Furthermore, due to the inherent randomness in its output, which results in variability with each generation, we believe it does not meet the requirements of industrial development so far.

### 4.4 User Study on Code Quality

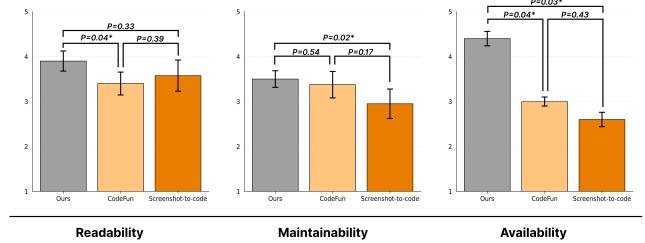


FIGURE 6: Results of User Study

Beyond evaluating the rendered visual effects, the overall quality of code is a crucial criterion for assessing the performance of automatic generation methods. In large-scale industrial development, code readability, availability, and maintainability are three key metrics for evaluation [12, 14]. Readability assesses whether the generated code is easy to understand, featuring a clear structure and using of meaningful names. Availability measures the extent to which the code requires manual modifications, calculated as:  $1 - \frac{\text{lines of code changes}}{\text{total lines of code}}$ . Meanwhile, maintainability evaluates the difficulty of maintaining the code.

For our user study, we invite five front-end engineers with an average work experience of over two years. Each participant is asked to review the code for 10 generated UI pages. For readability and maintainability, they report their assessments on a 5-point Likert scale. Regarding availability, participants are required to modify the code for each page until it matches basic business requirements. We used git to track code modifications and, following the method of Chen et al. [14], mapped the original scores starting from 0.8, with each 5% interval corresponding to a 1-5 scale.

We present the average scores in Fig. 6, with error bars indicating the standard error, and we report the p-values using the Mann-Whitney U test. For readability, our method achieves a score of 3.9, surpassing CodeFun’s 3.4 and Screenshot-to-code’s 3.58. Screenshot-to-code performs slightly better than CodeFun in terms of readability, primarily because it generates less code with a simpler structure. For the same reason, we did not find any significance in readability. In terms of maintainability, our method scores 3.5, also higher than CodeFun’s 3.38 and Screenshot-to-code’s 2.95. Significantly, a notable difference was observed between our method and Screenshot-to-code ( $p = 0.02^*$ ), with participants unanimously agreeing that our method performed better in responsive design compared to the others. No significant difference was observed between our method and CodeFun ( $p = 0.54$ ); a potential shortcoming of our method lies in its less effective reuse of certain container CSS properties compared to CodeFun. Finally, we observed significant differences in availability ( $p = 0.04^*$  with CodeFun and  $p = 0.03^*$  with Screenshot-to-code). Our method scores 4.4, significantly higher than CodeFun’s 3.0 and Screenshot-to-code’s 2.6. For our method, participants rarely needed to adjust the HTML structure and only had to make minor modifications to the CSS to meet business requirements.

## 5. DISCUSSION AND FUTURE WORK

In this paper, we aim to bridge the gap in generating front-end code from GUI design prototypes, implementing an end-to-end approach that encompasses five key steps: design prototypes linting, code layout tree construction, GUI element recognition, and code generation. Design prototypes linting is a critical step for automated code generation, especially for designs with quality deficiencies. Incorrect hierarchical structures [12–14] have been shown to adversely affect both the rendering effect and functionality of the generated code. In our approach, we primarily include detection of fragmented element layers and perceptual grouping processes. These two components together form an understanding of nested grouping within UI interfaces: starting from UI fragmented elements with incomplete semantics, moving to image-text groups with complementary semantics, and finally obtaining functional groups at a larger component scale. The UI structure generated on this basis is more organized and less prone to errors. The case2 in Fig. 5 offers an example. In the effect rendered by CodeFun, we can see some image and text elements were incompletely displayed, resulting in positional offsets. This occurred because some graphic-text elements were paired and placed under a single “div”, while others were wrapped in separate “divs,” leading to errors during rendering based on CSS properties. In our method, however, this part of the code contains three nested div layers: the innermost container wraps each image-text group, the second layer encloses a row of five icons, and the outermost layer contains two row elements. In such scenarios, CSS properties can be highly reusable and less prone to errors. For the fragmented elements detection module, as we directly used the same code and model weights from Li et al. [13], we did not further validate its effectiveness in our experiments. As for perceptual grouping, the current overall recognition accuracy has only reached 0.64. Continuing to improve the detection accuracy will be a direction for future efforts. Some potential strategies include incorporating larger Transformer models trained with a significant number of parameters, applying more UI data for specialized training, or making fuller use of multimodal UI data.

Through visual quality assessments, we compared the UI rendering effects produced by Prototype2Code with those of CodeFun and Screenshot-to-code. The results indicate that our method more faithfully reproduces design prototypes with fewer errors. From the perspective of large-scale development, both our method and CodeFun generally meet the needs for page effect realization. Screenshot-to-code, however, still has much room for improvement. Considering that the quality of the generated code cannot be fully assessed through visual effects alone, we recruited professional front-end engineers and sought their advice on the code’s readability, usability, and maintainability. Overall, the respondents believed that the code generated by our method performed best across all three dimensions. Simultaneously, based on the actual needs of large-scale industrial development, they proposed three feasible directions for future optimization. Firstly, the current method only supports the generation of code for static pages and cannot recognize the polymorphism of elements or support the interaction logic between elements. Specifically, for UI elements with multiple states, Prototype2Code can only convert them into images, which still requires manual modification. Sec-

ondly, Prototype2Code supports only the generation of native HTML+CSS code. It is crucial to support for popular front-end development frameworks such as Vue and React, as well as cross-platform code generation. Lastly, the current method lacks interaction design. Developing a visual platform similar to CodeFun, which supports personalized operations by engineers, is a potential area for future research.

## 6. CONCLUSION

In this paper, we introduce Prototype2Code, an end-to-end approach designed for generating front-end code directly from UI design prototypes. By integrating prototype linting into the code generation workflow, Prototype2Code reduces the impact of structural flaws in design prototypes on the generation outcomes, thereby enhancing the robustness of the generation process. The rendered UI effects from Prototype2Code demonstrate superior visual quality, achieving an SSIM of 0.91, a PSNR of 21.15, and an MSE of 0.97e-2, significantly surpassing other existing methods. Our user study on code quality further provides an assessment from a development perspective. Overall, the code generated by Prototype2Code, due to the clear structure offered by the layout tree and the semantics obtained from type recognition, surpasses other methods in readability. Owing to the nested grouping and CSS generated by LLM, it also exhibits better availability, requiring only minimal manual adjustments to meet basic business requirements. Additionally, its compatibility with responsive design principles contribute to improved maintainability.

## REFERENCES

- [1] Chen, Chunyang, Su, Ting, Meng, Guozhu, Xing, Zhenchang and Liu, Yang. “From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation.” *Proceedings of the 40th International Conference on Software Engineering*: pp. 665–676. 2018.
- [2] Deka, Biplob, Huang, Zifeng, Franzen, Chad, Hirschman, Joshua, Afergan, Daniel, Li, Yang, Nichols, Jeffrey and Kumar, Ranjitha. “Rico: A mobile app dataset for building data-driven design applications.” *Proceedings of the 30th annual ACM symposium on user interface software and technology*: pp. 845–854. 2017.
- [3] Li, Gang, Baechler, Gilles, Tragut, Manuel and Li, Yang. “Learning to denoise raw mobile UI layouts for improving datasets at scale.” *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*: pp. 1–13. 2022.
- [4] Mohian, Soumik and Csallner, Christoph. “Doodle2App: native app code by freehand UI sketching.” *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*: pp. 81–84. 2020.
- [5] Feng, Sidong, Yuan, Mingyue, Chen, Jieshan, Xing, Zhenchang and Chen, Chunyang. “Designing with Language: Wireframing UI Design Intent with Generative Large Language Models.” (2023). URL [arXiv2312.07755](https://arxiv.org/abs/2312.07755).
- [6] Chen, Sen, Fan, Lingling, Su, Ting, Ma, Lei, Liu, Yang and Xu, Lihua. “Automated cross-platform GUI code generation for mobile apps.” *2019 IEEE 1st International Workshop on*

- Artificial Intelligence for Mobile (AI4Mobile)*: pp. 13–16. 2019. IEEE.
- [7] Zhu, Lin, Feng, Yingchaojie, Zhu, Hang, Wang, Sijia, Zhu, Minfeng, Yu, Chenhao, Zhang, Yuhui, Xu, Daxing, Zhao, Deming, Feng, Yujun and Chen, Wei. “Figma2Code: Automatic code generation method for Figma design drafts.” *Journal of Computer-Aided Design & Computer Graphics*. (2023): pp. 1–9DOI [10.3724/SP.J.1089.2023-00336](https://doi.org/10.3724/SP.J.1089.2023-00336).
- [8] “Intelligent Code Generation for Design Drafts.” (2022). URL <https://www.imgcook.com/>.
- [9] “Codefun platform.” (2022). URL <https://code.fun/>.
- [10] “Sketch software.” (2022). URL <https://www.sketch.com/>.
- [11] “Figma platform.” (2023). URL <https://www.figma.com/>.
- [12] Chen, Yunnong, Zhen, Yankun, Shi, Chunling, Li, Jiazhi, Chen, Liuqing, Li, Zejian, Sun, Lingyun, Zhou, Tingting and Chang, Yanfang. “UI Layers Merger: merging UI layers via visual learning and boundary prior.” *Frontiers of Information Technology & Electronic Engineering* (2022): p. 373–387DOI [10.1631/FITEE.2200099](https://doi.org/10.1631/FITEE.2200099).
- [13] Li, Jiazhi, Zhou, Tingting, Chen, Yunnong, Chang, Yanfang, Zhen, Yankun, Sun, Lingyun and Chen, Liuqing. “ULDGNN: A Fragmented UI Layer Detector Based on Graph Neural Networks.” *arXiv preprint arXiv:2208.06658* (2022): p. 1–12URL <https://arxiv.org/abs/2208.06658>.
- [14] Chen, Liuqing, Chen, Yunnong, Xiao, Shuhong, Song, Yaxuan, Sun, Lingyun, Zhen, Yankun, Zhou, Tingting and Chang, Yanfang. “EGFE: End-to-end Grouping of Fragmented Elements in UI Designs with Multimodal Learning.” *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*: pp. 1–12. 2024.
- [15] Xiao, Shuhong, Zhou, Tingting, Chen, Yunnong, Zhang, Dengming, Chen, Liuqing, Sun, Lingyun and Yue, Shiyu. “UI Layers Group Detector: Grouping UI Layers via Text Fusion and Box Attention.” *CAAI International Conference on Artificial Intelligence*: pp. 303–314. 2022. Springer.
- [16] Xiao, Shuhong, Chen, Yunnong, Song, Yaxuan, Chen, Liuqing, Sun, Lingyun, Zhen, Yankun, Chang, Yanfang and Zhou, Tingting. “UI semantic component group detection: Grouping UI elements with similar semantics in mobile graphical user interface.” *Displays* Vol. 83 (2024): p. 102679.
- [17] “Figma to Code.” (2023). URL <https://www.figma.com/community/plugin/842128343887142055>.
- [18] “builder.io.” (2023). URL <https://www.builder.io/>.
- [19] “yotako.” (2023). URL <https://yotako.io/>.
- [20] Ikumapayi, Nathan Adelola. “Automated Front-End Code Generation Using OpenAI: Empowering Web Development Efficiency.” Available at SSRN 4590704 (2023): pp. 1–11.
- [21] Nam, Daye, Macvean, Andrew, Hellendoorn, Vincent, Vasilescu, Bogdan and Myers, Brad. “In-IDE Generation-based Information Support with a Large Language Model.” *arXiv preprint arXiv:2307.08177* (2023): pp. 1–13.
- [22] “Anima: AI Design to Code.” (2023). URL <https://www.animaapp.com/>.
- [23] Zhang, Xiaoyi, de Greef, Lilian, Swearngin, Amanda, White, Samuel, Murray, Kyle, Yu, Lisa, Shan, Qi, Nichols, Jeffrey, Wu, Jason, Fleizach, Chris, Everitt, Aaron and Bigham, Jeffrey P. “Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels.” *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*: pp. 1–15. 2021. Association for Computing Machinery, New York, NY, USA. DOI [10.1145/3411764.3445186](https://doi.org/10.1145/3411764.3445186).
- [24] Xie, Mulong, Xing, Zhenchang, Feng, Sidong, Xu, Xiwei, Zhu, Liming and Chen, Chunyang. “Psychologically-inspired, unsupervised inference of perceptual groups of GUI widgets from GUI images.” *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*: pp. 332–343. 2022.
- [25] He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing and Sun, Jian. “Deep residual learning for image recognition.” *Proceedings of the IEEE conference on computer vision and pattern recognition*: pp. 770–778. 2016.
- [26] Vu, Thang, Jang, Hyunjun, Pham, Trung X and Yoo, Chang. “Cascade rpn: Delving into high-quality region proposal network with adaptive convolution.” *Advances in neural information processing systems* Vol. 32 (2019): pp. 1–11.
- [27] Chen, Kai, Wang, Jiaqi, Pang, Jiangmiao, Cao, Yuhang, Xiong, Yu, Li, Xiaoxiao, Sun, Shuyang, Feng, Wansen, Liu, Ziwei, Xu, Jiarui et al. “MMDetection: Open mmlab detection toolbox and benchmark.” *arXiv preprint arXiv:1906.07155* (2019): pp. 1–13.
- [28] Chen, Jieshan, Xie, Mulong, Xing, Zhenchang, Chen, Chunyang, Xu, Xiwei, Zhu, Liming and Li, Guoqiang. “Object detection for graphical user interface: Old fashioned or deep learning or a combination?” *proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*: pp. 1202–1214. 2020.
- [29] Zang, Xiaoxue, Xu, Ying and Chen, Jindong. “Multimodal icon annotation for mobile applications.” *Proceedings of the 23rd International Conference on Mobile Human-Computer Interaction*: pp. 1–11. 2021.
- [30] Ying, Chengxuan, Cai, Tianle, Luo, Shengjie, Zheng, Shuxin, Ke, Guolin, He, Di, Shen, Yanming and Liu, Tie-Yan. “Do transformers really perform badly for graph representation?” *Advances in Neural Information Processing Systems* Vol. 34 (2021): pp. 28877–28888.
- [31] Rampášek, Ladislav, Galkin, Michael, Dwivedi, Vijay Prakash, Luu, Anh Tuan, Wolf, Guy and Beaini, Dominique. “Recipe for a general, powerful, scalable graph transformer.” *Advances in Neural Information Processing Systems* Vol. 35 (2022): pp. 14501–14515.
- [32] Gilmer, Justin, Schoenholz, Samuel S, Riley, Patrick F, Vinyals, Oriol and Dahl, George E. “Neural message passing for quantum chemistry.” *International conference on machine learning*: pp. 1263–1272. 2017. PMLR.
- [33] “Screenshot-to-Code.” (2024). URL <https://github.com/abi/screenshot-to-code/>.
- [34] “OpenAI Chat.” (2023). URL <https://chat.openai.com/>.
- [35] “Figma API.” (2023). URL <https://www.figma.com/developers/api/>.

- [36] Lin, Tsung-Yi, Maire, Michael, Belongie, Serge, Hays, James, Perona, Pietro, Ramanan, Deva, Dollár, Piotr and Zitnick, C Lawrence. “Microsoft coco: Common objects in context.” *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6–12, 2014, Proceedings, Part V* 13: pp. 740–755. 2014. Springer.
- [37] Ren, Shaoqing, He, Kaiming, Girshick, Ross and Sun, Jian. “Faster r-cnn: Towards real-time object detection with region proposal networks.” *Advances in neural information processing systems* Vol. 28 (2015): pp. 1–9.
- [38] Wang, Jiaqi, Chen, Kai, Yang, Shuo, Loy, Chen Change and Lin, Dahua. “Region proposal by guided anchoring.” *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*: pp. 2965–2974. 2019.
- [39] Duan, Kaiwen, Bai, Song, Xie, Lingxi, Qi, Honggang, Huang, Qingming and Tian, Qi. “Centernet: Keypoint triplets for object detection.” *Proceedings of the IEEE/CVF international conference on computer vision*: pp. 6569–6578. 2019.
- [40] Chen, Qiang, Wang, Yingming, Yang, Tong, Zhang, Xiangyu, Cheng, Jian and Sun, Jian. “You only look one-level feature.” *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*: pp. 13039–13048. 2021.
- [41] Zhu, Xizhou, Su, Weijie, Lu, Lewei, Li, Bin, Wang, Xiaogang and Dai, Jifeng. “Deformable detr: Deformable transformers for end-to-end object detection.” *arXiv preprint arXiv:2010.04159* (2020): pp. 1–16.
- [42] Manandhar, Dipu, Jin, Hailin and Collomosse, John. “Magic Layouts: Structural Prior for Component Detection in User Interface Designs.” *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*: pp. 15804–15813. 2021. DOI [10.1109/CVPR46437.2021.01555](https://doi.org/10.1109/CVPR46437.2021.01555).