AutoStructGUI: Bridging Design and Implementation of GUI through Structured Layout Generation Supplementary Material

Junquan Ren
College of Computer Science and Software Engineering
Shenzhen University
Shenzhen, China
junquan.dwade@gmail.com

Pengfei Xu*
College of Computer Science and Software Engineering
Shenzhen University
Shenzhen, China
xupengfei.cg@gmail.com

Abstract

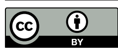
This supplementary material provides additional technical details and experimental evidence to support the main paper. First, it explains how predicted GUI layout sequences are converted back into hierarchical layout trees under both breadth-first and depth-first serialization schemes, with corresponding pseudocode. Second, it details the top-down refinement process, including overlap optimization and alignment optimization algorithms that improve layout validity and visual regularity. Third, it introduces the branch-switch algorithm, which adjusts ancestor-level child orderings to control depth-first serialization without altering the underlying tree structure. Fourth, it reports quantitative evaluations of structured layout quality using alignment and overlap metrics, analyzing the contribution of each refinement module. Fifth, it provides an ablation study that compares the DFS and BFS serialization schemes. Sixth, it describes the GUI development tasks and questionnaire design used in the user study. Seventh, it clarifies the code correctness and rendering reliability, highlighting the advantages of a structured intermediate representation and rule-based code generation. Eighth, it provides modeling details, including quantization, architecture, hyperparameters, and loss design, to ensure reproducibility. Finally, it presents the complete set of GUI results produced by all participants in Study I.

ACM Reference Format:
Junquan Ren and Pengfei Xu. 2026. AutoStructGUI: Bridging Design and Implementation of GUI through Structured Layout Generation Supplementary Material. In *31st International Conference on Intelligent User Interfaces (IUI '26), March 23–26, 2026, Paphos, Cyprus.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3742413.3789058

1 Conversion Of Structured GUI

A sequence of nodes, defined in a breadth-first or a depth-first way, can be transformed into a tree structure. We adopt queues for the breadth-first traversal and stacks for depth-first traversal.

*Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
IUI '26, Paphos, Cyprus
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1984-4/2026/03
https://doi.org/10.1145/3742413.3789058

1.1 Breadth-first Sequence Conversion

As illustrated in Algorithm 1, it starts with an initial dummy parent and two queues: $father_list$ representing the current level of potential parent nodes and $next_father_list$ for candidates at the next level. As each node is processed, it is attached to the first parent available from the current level. If the node is a branch node, it is added to the next-level candidate list; if it is the last child of its parent, that parent is removed immediately from the current list. When the current list is exhausted, the algorithm transitions to the next level of parents.

Algorithm 1 BFS_Conversion

```
1:  Input: prediction_list
2:  Output: root_node
3:
4:  fake_father ← {}
5:  father_list ← [fake_father]
6:  next_father_list ← [ ]
7:  for each new_node in prediction_list do
8:      if father_list is not empty then
9:          current_father ← father_list[0]
10:         new_node.father ← current_father
11:         Append new_node to current_father.children
12:     end if
13:     if new_node is a branch node then
14:         Append new_node to next_father_list
15:     end if
16:     if new_node is marked as the last child then
17:         Remove the first element from father_list
18:     end if
19:     if father_list is empty then
20:         father_list ← next_father_list
21:         next_father_list ← [ ]
22:     end if
23: end for
24: root_node ← prediction_list[0]
25: return root_node
```

1.2 Depth-first Sequence Conversion

As illustrated in Algorithm 2, it begins with a dummy root and uses a $father_stack$ to track potential parent nodes. For each node in the prediction list, the algorithm pops the current parent from the stack, assigns it as the parent of the new node, and appends the new node to the parent's children. If the new node is not the last

child, the current parent is pushed back onto the stack to allow for additional children. Furthermore, if the new node is capable of having descendants (i.e., it is not a leaf), it is pushed onto the stack as a candidate for future parenthood. This process continues until the stack is empty, resulting in a fully constructed tree rooted in the first element of the prediction list.

Algorithm 2 DFS_Conversion

```
1:  Input: prediction_list
2:  Output: root_node
3:
4:  fake_father ← {}
5:  father_stack ← [fake_father]
6:  pred_step ← 0
7:  while father_stack is not empty do
8:      current_father ← Pop(father_stack)
9:      new_node ← prediction_list[pred_step]
10:     pred_step ← pred_step + 1
11:     new_node.father ← current_father
12:     Append new_node to current_father.children
13:     if new_node is not the last child then
14:         Push(current_father, father_stack)
15:     end if
16:     if new_node is not a leaf node then
17:         Push(new_node, father_stack)
18:     end if
19: end while
20: root_node ← prediction_list[0]
21: return root_node
```

2 Details Of Top-down Refinement

2.1 Overlap Optimization

For potential overlap issues, we optimize them in a single traversal of the layout tree, as illustrated in Algorithms 3, 4, 5 and 6. For any layout tree L, we perform a top-down breadth-first traversal. For any node N_i that has children, we iteratively adjust these child nodes subject to the constraint of N_i's boundary.

In each iteration, we compute the necessary horizontal and vertical separation adjustment vectors for every pair of overlapping bounding boxes. Then, we test whether the adjusted bounding boxes, when moved exclusively in the horizontal direction or exclusively in the vertical direction, remain within N_i's boundary. If both adjustments are valid, we choose the one with the smaller magnitude; if only one is valid, we adopt the valid adjustment; otherwise, the original position is maintained. The iteration ends when all candidate adjustments fall below a preset threshold.

Subsequently, for each child node N_j of N_i, we calculate the offset (dx, dy) between the positions before and after the adjustment and propagate this offset to the subtree of N_j, ensuring that all nodes in the subtree maintain consistent offsets. We apply this process to every node during the traversal, thus optimizing overlapping issues throughout the entire layout tree L while preserving the relative positional information of overlapping nodes.

Algorithm 3 Tree Overlap Processing

```
1:  Input: node, margin
2:
3:  if node has children then
4:      father_box ← node.abs_box
5:      for each child of node do
6:          record original position
7:      end for
8:      if multiple children then
9:          RemoveOverlapsWithMargin(child_boxes, father_box,
            margin)
10:     end if
11:     for each child of node do
12:         if child's box moved then
13:             PropagateAdjustment(child, dx, dy)
14:         end if
15:     end for
16:     for each child of node do
17:         ProcessTreeOverlap(child, margin)
18:     end for
19: end if
```

Algorithm 4 Propagate Adjustment

```
1:  Input: node, dx, dy
2:
3:  Add (dx, dy) to node.abs_box
4:  for each child of node do
5:      PropagateAdjustment(child, dx, dy)
6:  end for
```

Algorithm 5 Remove Overlaps with Margin

```
1:  Input: boxes, father_box, margin
2:  Output: boxes
3:
4:  for a fixed number of iterations do
5:      for each unique pair (boxA, boxB) in boxes do
6:          (dx, dy) ← SeparationAdjustment(boxA, boxB, margin)
7:          accumulate adjustments
8:      end for
9:      if adjustments are below threshold then
10:         return boxes
11:     end if
12:     for each box in boxes do
13:         apply valid adjustment (horizontal OR vertical)
14:             that keeps box inside father_box and [0, 1]
15:     end for
16: end for
17: return boxes
```

2.2 Align Optimization

Specifically, the algorithm performs a top-down breadth-first traversal of the layout tree L, as illustrated in Algorithms 7, 8, 9, 10 and 11. For any node N_i with child nodes, the algorithm pairs its child nodes, calculates the horizontal and vertical spacings separately,

Algorithm 6 Separation Adjustment

```
1:  Input: boxA, boxB, margin
2:  Output: (dx, dy)
3:
4:  unpack [x, y, w, h] for boxA and boxB
5:  compute overlap_x, overlap_y
6:  deficiency_x ← 0, deficiency_y ← 0
7:  if overlap_x > 0 and overlap_y > 0 then
8:      deficiency_x ← margin + overlap_x
9:      deficiency_y ← margin + overlap_y
10: end if
11: tol ← 1 × 10^-9
12: if boxA_w > 1 and boxB_w > 1 then
13:     lock horizontal adjustment
14: end if
15: if boxA_h > 1 and boxB_h > 1 then
16:     lock vertical adjustment
17: end if
18: compute candidate (dx, dy) based on box centers
19: return (dx, dy)
```

and determines the amount of overlap in each direction (horizontal and vertical) based on a predefined threshold. When two nodes have sufficient overlap in one direction and the spacing in the other direction is within the threshold (but not overlapping), the alignment condition is considered satisfied. At this point, we further compare the differences in spacing between the boundaries of the nodes and select the optimal alignment method.

Through this step-by-step local alignment strategy, the algorithm ultimately optimizes all nodes in the layout tree, resulting in a neater and visually pleasing interface layout.

Algorithm 7 Adjust Layout Align

```
1:  Input: data
2:  Output: data
3:
4:  Initialize a FIFO queue with (data, 0)
5:  while queue is not empty do
6:      Dequeue (node, level)
7:      if node has children then
8:          for each pair (child_i, child_j) with i < j do
9:              adjust child_j.abs_box using
10:                 CalculateAlign(child_i.abs_box, child_j.abs_box, node.abs_box)
11:         end for
12:         Enqueue all children with level + 1
13:     end if
14: end while
15: return data
```

3 Details Of Branch Switch Algorithm

The branch-switch algorithm operates by iteratively modifying ancestor-level child orderings along the path from a selected node to the root, as illustrated in Algorithm 12. The algorithm first locates the selected node in the node list. It then enters a loop that climbs the tree via parent pointers. At each iteration, the current node is removed from its parent's children list and appended to the end of that list. This single operation guarantees that, under depth-first traversal, the entire subtree rooted at the current node will be visited after all its siblings. The algorithm then promotes the parent as the new current node and repeats the same reordering step. By propagating this operation upward, the selected branch is consistently pushed to the end at every ancestor level, ensuring that the selected subtree becomes last in the overall DFS serialization, while leaving all node attributes and the tree topology unchanged.

Algorithm 8 Get X Distance

```
1:  Input: bbox1, bbox2
2:  Output: horizontal distance
3:
4:  each bbox is [x, y, w, h]
5:  if bbox2 is completely to the left of bbox1 then
6:      return the horizontal gap between them
7:  else if bbox2 is completely to the right of bbox1 then
8:      return the horizontal gap between them
9:  else
10:     return 0
11: end if
```

Algorithm 9 Get Y Distance

```
1:  Input: bbox1, bbox2
2:  Output: vertical distance
3:
4:  each bbox is [x, y, w, h]
5:  if bbox2 is completely above bbox1 then
6:      return the vertical gap between them
7:  else if bbox2 is completely below bbox1 then
8:      return the vertical gap between them
9:  else
10:     return 0
11: end if
```

4 Structured Layout Quality Metrics

We conducted an experiment to evaluate the quality of GUI generation by our model and assess the effect of our Top-down refinement on improving GUI layout quality. Since existing data-driven methods cannot generate structured GUI layouts, we did not compare our model with them.

During training, we used 95% of the GUIs from the MUD dataset [1] as the training set (a total of 13,268 GUIs). To evaluate the performance of our model, the remaining 5% of the GUI layouts were used as the test set, totaling 1,507 GUIs. We adopt the alignment score (**Align**) [3] and the overlap score (**Overlap**) [4]. We extract type conditions from the test set as input, use our model to generate layouts corresponding to these conditions, and compute the metrics. Meanwhile, we also calculate these two metrics on the test set layouts for algorithmic comparison; the results are shown in Table 1. Here, "Ground Truth" represents the metrics on the test set, "w/o overlap" and "w/o align" refer to the metrics of our

Algorithm 10 Calculate Align

```
1: Input: bbox1, bbox2, father_box, box_xy_is_center (default: FALSE)
2: Output: (bbox2.x, bbox2.y)
3:
4: check_time ← 5
5: (do_align_x, do_align_y) ← CheckAlignType(bbox1, bbox2)
6: while do_align_x ∧ do_align_y ∧ (check_time > 0) do
7:    Try to remove overlap by slight adjustments
8:    (bbox1, bbox2) ← RemoveOverlaps([bbox1, bbox2], father_box, margin = 0.03)
9:    (do_align_x, do_align_y) ← CheckAlignType(bbox1, bbox2)
10:   check_time ← check_time − 1
11: end while
12: if do_align_x then
13:   if left alignment error is smallest then
14:     new_x ← bbox1.x
15:   else if center alignment error is smallest then
16:     new_x ← bbox1.x + (bbox1.width − bbox2.width)/2
17:   else
18:     new_x ← bbox1.x + bbox1.width − bbox2.width
19:   end if
20:   Apply new_x if within father's bounds
21:   if new_x is within father_box horizontally then
22:     bbox2.x ← new_x
23:   end if
24: end if
25: if do_align_y then
26:   if top alignment error is smallest then
27:     new_y ← bbox1.y
28:   else if center alignment error is smallest then
29:     new_y ← bbox1.y + (bbox1.height − bbox2.height)/2
30:   else
31:     new_y ← bbox1.y + bbox1.height − bbox2.height
32:   end if
33:   Apply new_y if within father's bounds
34:   if new_y is within father_box vertically then
35:     bbox2.y ← new_y
36:   end if
37: end if
38: return (bbox2.x, bbox2.y)
```

Algorithm 11 Check Align Type

```
1: Input: bbox1, bbox2
2: Output: (do_align_x, do_align_y)
3:
4: x_dist ← GetXDistance(bbox1, bbox2)
5: y_dist ← GetYDistance(bbox1, bbox2)
6: do_align_x ← (bbox1 and bbox2 overlap horizontally) and (y_dist ≤ 0.05)
7: do_align_y ← (bbox1 and bbox2 overlap vertically) and (x_dist ≤ 0.25)
8: return (do_align_x, do_align_y)
```

Algorithm 12 Branch Switch Algorithm

```
1: Input: node_list
2:
3: selected_node ← null
4: for each node in node_list do
5:   if node has key 'selected' and node['selected'] =' selected' then
6:     selected_node ← node
7:     exit loop
8:   end if
9: end for
10:
11: continue ← true
12: while continue do
13:   if selected_node has no key 'father' then
14:     continue ← false
15:   else
16:     father ← selected_node.father
17:     child_list ← father.children
18:     remove selected_node from child_list
19:     append selected_node to child_list
20:     selected_node ← father
21:   end if
22: end while
```

Table 1: Element metrics

	Align ↓	**Overlap ↓**
ours(w/o overlap)	**0.0013**	0.061
ours(w/o align)	0.0025	0.048
ours	<u>0.0016</u>	<u>0.046</u>
Ground Truth	0.0018	**0.015**

model without overlap optimization and alignment optimization. respectively, while "ours" indicates our complete model.

The quantitative results show that removing the overlap module yields the best alignment score, but it leads to a larger overlap error; whereas removing the align module significantly reduces the alignment quality. It is important to note that in Android components, FrameLayout supports overlapping; therefore, the layouts generated by the model may also exhibit overlapping. Moreover, since our overlap optimization does not disrupt the original relative positions between nodes and uses the boundaries of the parent node as a constraint for offsets, it does not completely eliminate overlap. The complete model achieves a good balance between the two metrics, with both optimization modules functioning properly, resulting in a final layout that is both stable and aesthetically pleasing.

5 Ablation Study

To further illustrate why two serialization schemes are required for both global and local generation, we conduct an ablation study. On the MUD dataset split, we adopt the same experimental settings as those described in Section 4.

5.1 Settings

5.1.1 Evaluation metrics. For the global generation task, we adopt the Align score (**Align**) [3] and the overlap score (**Overlap**) [4] as element metrics to evaluate the quality of generated GUI components. In addition, to assess the quality of the generated layout structures, we introduce **S-Align** and **S-Overlap**, as defined in StructLayoutFormer [2], as structure metrics.

For the local generation task, since it is restricted to a local container, the structure metrics designed for global generation are no longer applicable. Therefore, we use **Align** and **Overlap** as element-level metrics. In addition, we define **Cross-Align** to measure the cross-container alignment consistency of components between the selected container and its sibling containers. Given a container node N_i marked as selected, let S denote the set of all elements in the subtree of N_i, and let B denote the set of all elements in the subtrees of the sibling container nodes of N_i. For each element $s \in S$, we compute its minimum alignment distance to the elements in B, the alignment distance is defined by the existing metric **Align**. Based on this, **Cross-Align** applies a negative logarithmic transformation to the alignment distance of each element and averages the results over all elements in S, which is defined as equation 1.

$$\text{Cross-Align} = \frac{1}{|S|} \sum_{s \in S} -\log\big(1 - \min_{b \in B} \mathrm{d}(s, b)\big). \tag{1}$$

5.1.2 Tasks. For the global generation task, we serialize the layout trees using BFS and DFS respectively and train two corresponding models. After performing global conditional generation on the test set, we compute the evaluation metrics based on the generated results.

For the local generation task, we first preprocess the test set before feeding it into the model. Specifically, given a layout tree L, we randomly select a container node N_c. After collecting the component conditions C from all components in the subtree rooted at N_c, we remove these components from the subtree. We then mark N_c as selected. At this point, the modified layout tree L together with the condition set C can be used as inputs for local generation. During training, we again employ two serialization schemes and enable the Random Swapping Strategy for both models. During inference, we uniformly apply the Branch Switch Strategy to the input layout tree L before passing it to the model. Given the existing "loss of context" problem, to adapt the BFS-based model to the local generation task, we mask nodes that violate the autoregressive causality constraint, as illustrated in Section 5.3 of the paper, and restore these nodes after the generation process is completed.

For both types of tasks, Top-down Refinement is disabled to eliminate potential confounding factors.

5.2 Results

The results in Tables 2 and 3 indicate that BFS and DFS serialization exhibit stable and complementary performance characteristics across global and local generation tasks.

In the global generation task, BFS significantly outperforms DFS on the **Overlap** and **S-Overlap** metrics, suggesting that breadth-first serialization is more conducive to generating structurally reasonable layouts. This advantage helps reduce element overlap and improves global consistency. Although DFS achieves a slightly better **Align** metric, its inferior performance on structure-related metrics indicates that relying solely on depth-first serialization is insufficient to ensure structural stability in layout generation.

Table 2: Quality metrics of DFS and BFS serialization under global generation.

Schemes	Element metrics		Structure metrics	
	Align ↓	Overlap ↓	S-Align ↓	S-Overlap ↓
BFS	0.0025	**0.075**	**0.0028**	**0.1200**
DFS	**0.0023**	0.1000	0.0029	0.1580

Table 3: Element metrics of DFS and BFS serialization under local generation.

Schemes	Element metrics		
	Cross-Align ↓	Align ↓	Overlap ↓
BFS	0.0548	0.0116	0.0440
DFS	**0.0286**	**0.0035**	**0.0280**

In the local generation task, DFS consistently outperforms BFS on **Cross-Align**, **Align**, and **Overlap**, demonstrating its clear advantage in local generation scenarios. This result directly reflects the "loss of context" issue encountered by BFS under autoregressive causal constraints, which substantially degrades generation quality. By contrast, DFS naturally preserves the complete layout context, making it better suited for local generation tasks.

In summary, BFS is more appropriate for global generation tasks, where maintaining overall structural coherence is critical, whereas DFS is better suited for local generation tasks, enabling high-quality and controllable local refinements while preserving full contextual information.

6 EVALUATION STUDY MATERIALS

6.1 Developing Tasks

6.1.1 User Settings. Develop a settings menu that provides users with a convenient way to configure key application parameters such as sound, brightness, and notifications. The interface should be organized, visually clear, and easy to navigate, enabling users to quickly locate specific options and make adjustments without confusion.

Component Requirements.

(1) **TextView**: Present descriptive labels and short explanations for each setting item, e.g., "Sound," "Brightness," or "Notifications," to ensure clarity of purpose.
(2) **ImageView**: Display the user's avatar or profile image to enhance personalization and visual recognition.
(3) **CheckBox / CheckTextView / Switch**: Provide simple toggles for binary options such as enabling or disabling notifications, activating night mode, or turning on vibration feedback.

(4) **SeekBar**: Allow fine-grained control for continuous values like volume or screen brightness through a draggable slider.
(5) **Spinner**: Offer drop-down lists for multi-choice configurations such as selecting language, region, or theme preferences.
(6) **EditText**: Enable direct numeric or textual input when precise customization is required, such as entering a specific brightness percentage.
(7) **Button**: A dedicated "Save Settings" action button that confirms the user's selections and applies changes immediately to the system.

6.1.2 User Login. Create a basic login interface that allows users to enter their account credentials and access the application by pressing the login button. The page should be simple, clear, and efficient to ensure smooth user interaction.

Component Requirements.

(1) **TextView**: Display labels or prompts such as "Account" and "Password" to guide user input.
(2) **EditText**: Provide input fields for entering the username and password.
(3) **Button**: Serve as the "Login" button that triggers the authentication process.
(4) **CheckBox**: Offer a "Remember Password" option with selectable state.
(5) **Switch / CheckBox**: Optionally allow users to toggle the "Show Password" feature.
(6) **ImageView**: Display the application logo or branding element for visual identity.

6.1.3 Shopping. Design a product detail page that thoroughly presents a single item's information, including images, videos, descriptions, and user reviews, while supporting purchase actions such as "Buy Now" and "Add to Cart." The layout should be clear and scannable, with smooth navigation across content sections.

Component Requirements.

(1) **ImageView**: Display high-resolution product images to showcase details.
(2) **ScrollView**: Enable vertical scrolling to reveal image galleries or long content.
(3) **TextView**: Show product name, detailed description, price, and stock status.
(4) **RatingBar**: Present average rating and optionally accept user ratings; surface review information.
(5) **Button**: Provide "Buy Now" and "Add to Cart" actions.
(6) **EditText**: Allow entry of purchase quantity, coupon codes, or optional notes.
(7) **Spinner**: Let users select product variants such as color, size, or model.

6.1.4 Review Menu. Build a simple review menu for evaluating a trip's itinerary, covering overall experience and itinerary arrangement. Provide quick-rating components first and clear entry points to detailed review pages, then allow submission. The interface must be intuitive and easy to operate.

Component Requirements.

(1) **TextView**: Display the page title. Show trip information such as itinerary name and date. Create small section headers for "Overall Experience" and "Itinerary Arrangement."
(2) **ImageView**: Present a promotional image of the trip.
(3) **RatingBar**: Allow quick star-based scoring for the "Overall Experience" item and visualize the selected score.
(4) **EditText**: Provide a text box for quick comments on the "Itinerary Arrangement" item to capture opinions or suggestions.
(5) **Button**: Provide an entry button for each item ("Overall Experience," "Itinerary Arrangement") to open the detailed review page. Include a final "Submit" button to send the evaluation.

6.2 Questionnaire

Below, we list the questions we used in the evaluation study questionnaire.

6.2.1 NASA-TLX. (Likert scale: 1 = Very low, 5 = Very high)

(1) How mentally demanding was the task?
(2) How physically demanding was the task?
(3) How hurried or rushed was the pace of the task?
(4) To what extent do you feel that your performance during the task was unsatisfactory or that you did not fully accomplish the task goals?
(5) How hard did you have to work to accomplish your level of performance?
(6) How insecure, discouraged, irritated, stressed, and annoyed were you?

6.2.2 System Usability Scale (SUS). (Likert scale: 1 = Strongly disagree, 5 = Strongly agree)

(1) I think that I would like to use this system frequently.
(2) I found the system unnecessarily complex.
(3) I thought the system was easy to use.
(4) I think that I would need the support of a technical person to be able to use this system.
(5) I found the various functions in this system were well integrated.
(6) I thought there was too much inconsistency in this system.
(7) I would imagine that most people would learn to use this system very quickly.
(8) I found the system very cumbersome to use.
(9) I felt very confident using the system.
(10) I needed to learn a lot of things before I could get going with this system.

7 Code Correctness and Rendering Reliability

In our framework, the learning model is responsible for producing a structured GUI layout representation, including the hierarchical tree structure and normalized layout attributes. The final Android XML code is then exported through a deterministic, rule-based code generation pipeline. As a result, the model does not directly produce XML syntax, and code correctness is primarily governed by the structural validity of the generated layout.

Regarding XML validity, as long as the generated structured layout satisfies basic structural constraints (e.g., a single root node,

valid parent-child relationships, and complete layout attributes), the exported XML code is syntactically valid and can be correctly parsed by Android Studio. During training data optimization, we rigorously filtered out the erroneous cases from the training set. In our experiments, we did not observe syntax-level XML errors such as malformed tags, invalid attributes, or unparsable layout files. This design intentionally separates structural generation from code realization, ensuring that syntactic correctness is guaranteed by construction.

Layout render failures are related to layout semantics, such as insufficient layout constraints, component attributes and deeply nested hierarchies. First, during training data optimization, we applied the same rigorous filtering to erroneous component attributes in the training set, such as invalid position or size values and overlapping or redundant layout hierarchies, in order to minimize the occurrence of such cases. Regarding insufficient layout constraints, this issue does not arise in our current setting, as the layout containers and all components used in this work are mutually compatible. Finally, when the model is integrated into the interactive system, each generated result undergoes a lightweight component attribute validation to ensure legality. In the rare cases where errors occur, the system either applies corrective adjustments, for example clamping a normalized horizontal coordinate from -0.1 to 0.0 when the valid range is [0.0, 1.0], or triggers regeneration by invoking the model again.

Overall, the use of a structured intermediate representation combined with rule-based code export provides AutoStructGUI with a robust advantage in code correctness and reliability, distinguishing it from end-to-end design-to-code approaches that directly generate executable layout code.

8 Modeling Details

Following prior work, we adopt the same quantization strategy, hyperparameter configuration, and loss design as StructLayoutFormer [2]. $[x_i, y_i, w_i, h_i]$, are first normalized to [0,1] and then uniformly quantized into discrete bins, enabling autoregressive prediction with categorical distributions. Element types t_i and structural indicators b_i^1, b_i^2 are represented as discrete tokens, consistent with the structure serialization scheme.

Our model architecture and training hyperparameters follow StructLayoutFormer. All Transformer modules use an embedding dimension of 512 and a feed-forward dimension of 2048. The conditional layout generator consists of 6 Transformer layers, while the context encoder contain 4 layers. We train the model using the Adam optimizer with the same learning rate and optimization settings reported in StructLayoutFormer.

Regarding the training objective, since all layout attributes are discretized, we employ cross-entropy losses for predicting node attributes and structural indicators. No additional loss terms or re-weighting strategies are introduced.

9 User Study I Result Images

In Study I, we recruited 8 participants, each of whom completed 6 GUI development tasks, yielding a total of 48 resulting GUIs. In this section, we present all the resulting GUIs in two figures, as shown in Figure 1 and Figure 2. Each row corresponds to all the results from one participant, with the 6 columns representing the GUIs created by each participant using our framework and the baseline for three tasks (User Login, Shopping, and Review Menu).

Licenses and Attributions

Some of the icons used in the figures of the paper (e.g., user avatar, thought bubble) are sourced from Google and are licensed under the Apache License, Version 2.0. © 2013 Google, Inc. Please see http://www.apache.org/licenses/LICENSE-2.0 for details.

Some of the icon assets used in the experiment of this work were sourced from the open source community and are used under the MIT License. The original copyright of these icons belongs to their respective authors, and we express our gratitude for their contributions. The MIT License permits copying, modification, and distribution, provided that the original copyright notice and license terms are retained. For more details, please refer to: https://opensource.org/licenses/MIT.

References

[1] Sidong Feng, Suyu Ma, Han Wang, David Kong, and Chunyang Chen. 2024. Mud: Towards a large-scale and noise-filtered ui dataset for modern style ui modeling. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–14.

[2] Xin Hu, Pengfei Xu, Jin Zhou, Hongbo Fu, and Hui Huang. 2025. StructLayoutFormer: Conditional Structured Layout Generation via Structure Serialization and Disentanglement. *IEEE Transactions on Visualization and Computer Graphics* (2025).

[3] Zhaoyun Jiang, Jiaqi Guo, Shizhao Sun, Huayu Deng, Zhongkai Wu, Vuksan Mijovic, Zijiang James Yang, Jian-Guang Lou, and Dongmei Zhang. 2023. Layoutformer++: Conditional graphic layout generation via constraint serialization and decoding space restriction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 18403–18412.

[4] Jianan Li, Jimei Yang, Jianming Zhang, Chang Liu, Christina Wang, and Tingfa Xu. 2020. Attribute-conditioned layout gan for automatic graphic design. *IEEE Transactions on Visualization and Computer Graphics* 27, 10 (2020), 4039–4048.

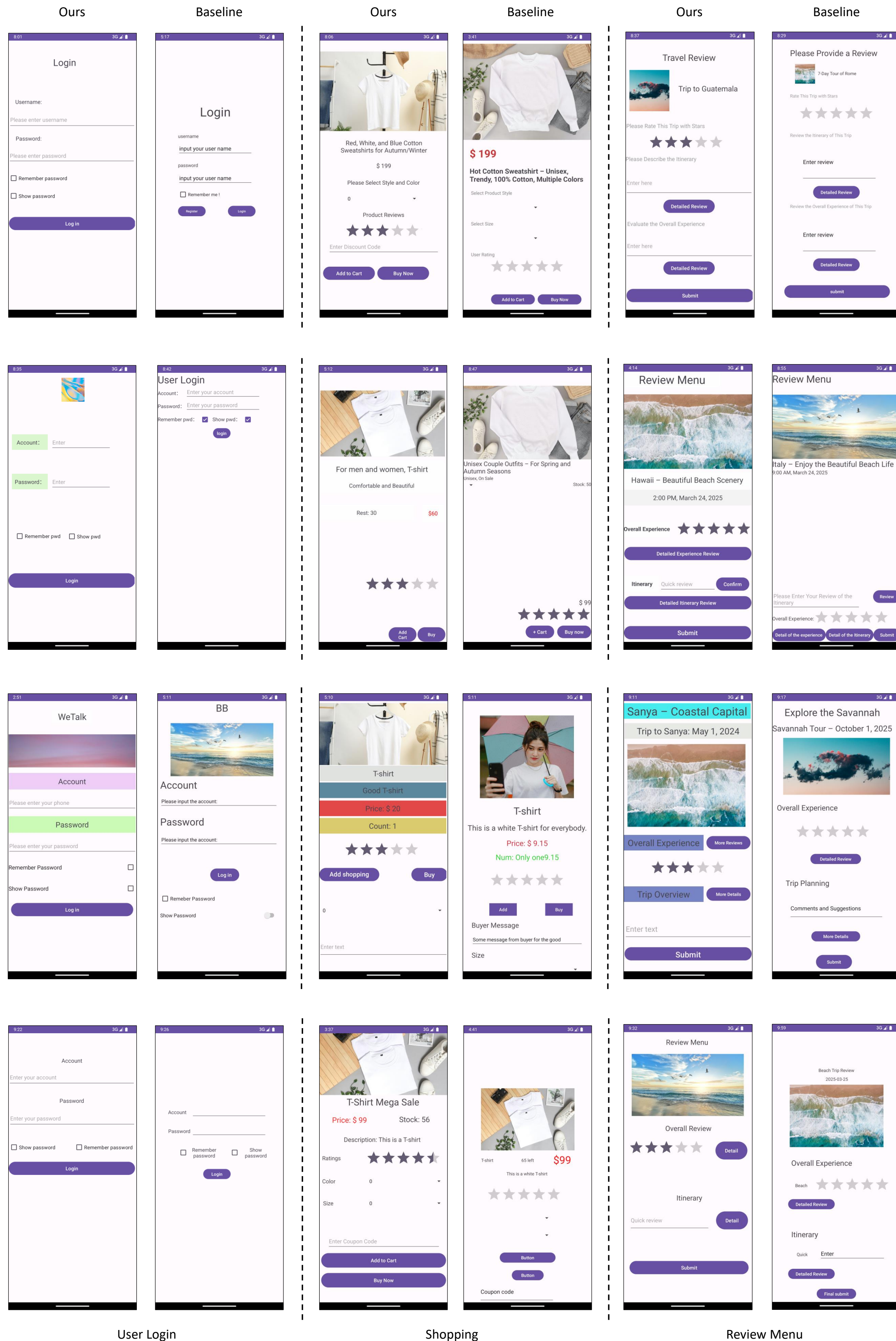


Figure 1: All results GUIs for the first four participants.

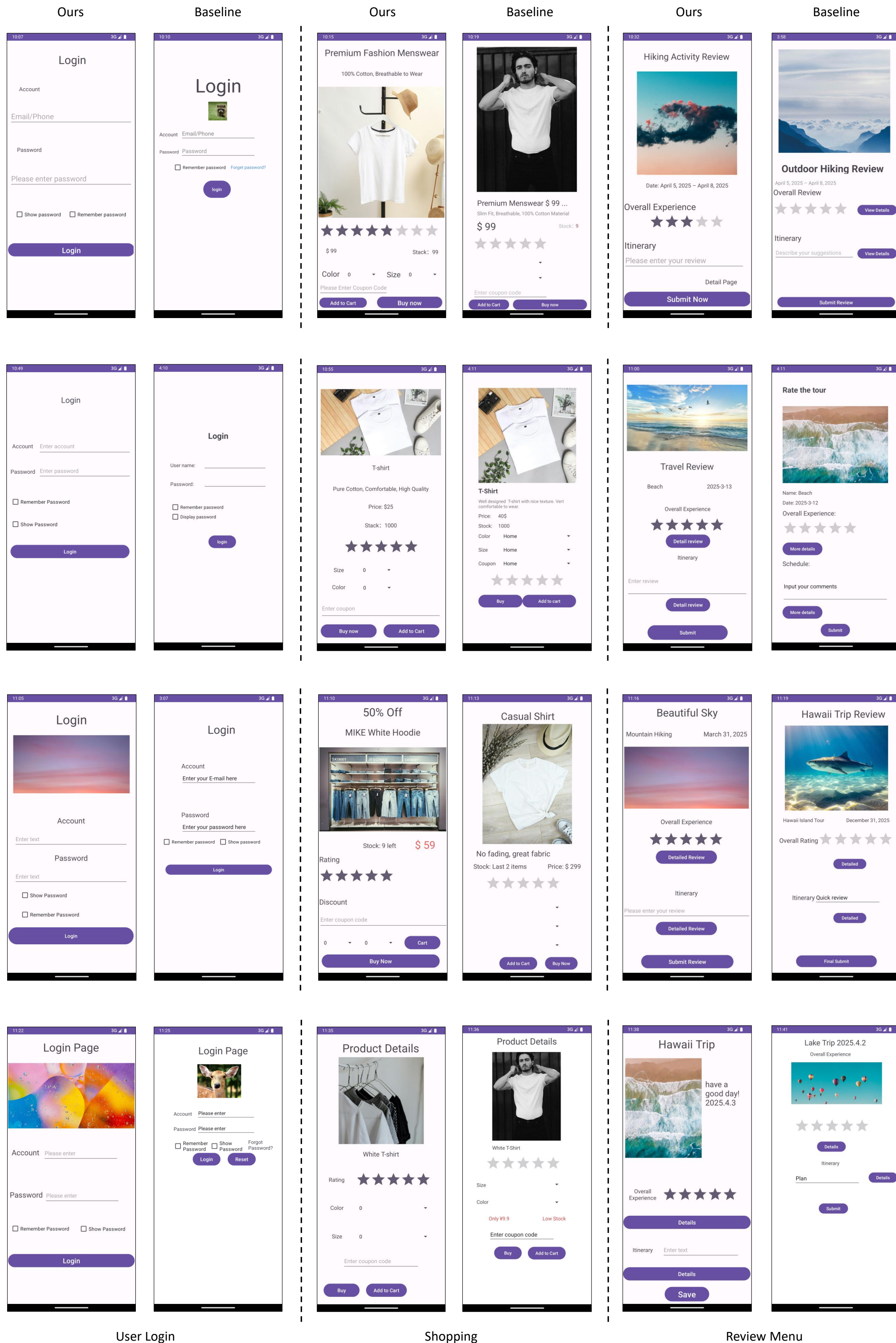


Figure 2: All results GUIs for the last four participants.