LayCoder: UI Layout Completion with an Encoder-Only Transformer and Layout Tokenizer

Iskandar Salama, Luiz Henrique Mormille, Masayasu Atsumi
Soka University, Tokyo, Japan

***Abstract*—The growing complexity of user interface (UI) design calls for effective methods to understand, complete, and refine layout structures. While prior work has focused predominantly on generating UI layouts from scratch, completing partially designed interfaces is equally critical, particularly in iterative design workflows and scenarios involving incomplete prototypes. In this study, we address the UI layout completion task for mobile app screens using an encoder-only transformer architecture with masked modeling and a layout tokenizer. By representing UI elements as discrete tokens, we formulate layout completion as a sequence prediction problem that leverages global context to infer missing components. We evaluate our approach on subsets of the RICO dataset designed with varying constraints on UI element types and spatial overlap, and report results using standard layout metrics: Coverage, Intersection over Union (IoU), Max IoU, and Alignment. The experiments demonstrate that the proposed method achieves substantial improvements over LayoutFormer++, a widely adopted baseline in UI layout generation, particularly in Coverage, and in several cases, IoU, Max IoU, and Alignment. Additional experiments on noise-reduced subsets reveal that dataset curation can enhance spatial consistency but may also reduce Coverage, reflecting an inherent trade-off between completeness and structural precision. These findings highlight both the effectiveness and the limitations of encoder-only masked modeling for layout completion, and underscore the importance of balancing model design with dataset construction when tackling complex UI design tasks.**

***Keywords*—Layout completion; deep learning; encoder-only transformers; masked language modeling; tokenization**

I. Introduction

In the ever-evolving landscape of mobile applications, user interfaces (UIs) play a pivotal role in determining user engagement and satisfaction. As mobile apps have evolved, the design of user interfaces has become more sophisticated, focusing on enhancing user experience through intuitive and visually appealing layouts. This shift towards more user-centric designs matters significantly because studies have shown that user-friendly interfaces can lead to better user retention and increased app usage [1].

Consequently, the demand for efficient and innovative UI design techniques has increased, prompting the integration of generative AI (Gen-AI) into the design process [2], [3]. Generative models offer substantial benefits in UI development, including improved speed and automation [4], reducing the time and effort required for manual layout design.

Historically, research in automatic UI layout modeling has focused primarily on generating complete layouts from scratch [5]. However, the ability to complete partially designed interfaces is equally important—particularly in scenarios that involve iterative refinement or require automatic completion based on existing components [6].

Formally, we define UI layout completion as a conditional structured prediction problem under partial observability. Given a partially observed layout L_{obs}, where a subset of UI elements are missing or masked, the objective is to predict the missing elements L_{miss}. The completed layout $\hat{L}$ should remain spatially and semantically consistent with realistic UI design patterns.

This task differs from unconditional layout generation, as the model must preserve existing components while inferring plausible missing structure conditioned on the visible layout context. This formulation reflects practical early-stage design scenarios, where interfaces are incomplete and require context-aware completion to support iterative refinement.

To address this problem, we investigate the potential of encoder-only transformer models to infer missing components from partial UI layouts. Transformers [7] have become a powerful tool across multiple domains, notably in natural language processing and, more recently, layout modeling tasks. Their ability to model long-range dependencies and process sequences in parallel makes them particularly effective for capturing the structure of complex inputs such as UI layouts.

Encoder-only transformers are particularly well-suited for tasks requiring contextual understanding. In layout completion, they can process all visible components and their interrelationships in a single pass, enabling the model to predict missing elements based on global layout context. This capability aligns closely with masked language modeling (MLM) [8], where missing tokens are reconstructed from surrounding information.

Although our formulation is inspired by MLM, LayCoder is not a straightforward adaptation of BERT to discretized layout tokens. Unlike natural language, UI layouts are inherently spatial and structured, requiring joint reasoning about geometry, element semantics, and layout coherence. In particular, each UI component cannot be represented as a single token, but rather as a set of correlated spatial and semantic attributes.

To enable Transformer-based modeling of UI layouts, we therefore convert each layout into a structured sequence of discrete tokens using a custom tokenization mechanism, which we term the *Layout Tokenizer*. The first token represents the *layout category*, and each UI element is encoded through multiple attribute tokens—namely, *centerX*, *centerY*, *width*, *height*, and the *component label*. These attributes are individually mapped into discrete token IDs from a predefined vocabulary.

This representation allows LayCoder to perform masked

completion at the level of structured UI elements, where each missing component corresponds to multiple correlated spatial and semantic attributes. This element-level grouping distinguishes LayCoder from standard MLM models by enabling masked prediction over multi-attribute UI components rather than single discrete symbols.

The Layout Tokenizer operates on a predefined vocabulary consisting of 675 tokens. This layout-specific tokenization transforms the structured UI into a flat sequence of tokens, enabling Transformer models to process and complete UI layouts in a manner similar to how language models predict missing words [9].

Our approach is evaluated using the RICO dataset [25], one of the most comprehensive resources for mobile UI layouts. It contains over 66,000 UI screens and more than 3 million annotated elements, making it a suitable benchmark for layout modeling and completion.

Our main contributions are summarized as follows:

- We propose LayCoder, an encoder-only transformer framework that formulates UI layout completion as masked structured prediction over partially observed mobile app layouts.
- We introduce a custom Layout Tokenizer, which represents UI elements as discrete sequences of spatial and semantic tokens. This design enables Transformer models to leverage global context for inferring missing components in incomplete layouts.
- Through experiments on subsets of the RICO dataset, we demonstrate that the proposed method achieves substantial improvements over LayoutFormer++ [24], a previously leading approach in UI layout generation, with consistent gains in Coverage and notable improvements in IoU and Alignment across multiple test scenarios. The empirical results underscore the efficacy of encoder-only masked modeling in capturing spatial and semantic dependencies within layout structures.
- We further analyze the role of dataset curation by evaluating on noise-reduced subsets. The findings reveal that filtering improves spatial consistency but can reduce Coverage, highlighting the observed trade-off between layout completeness (Coverage) and structural precision (IoU and Alignment) in UI layout completion.

II. Related Work

The automated modeling of UI layouts has attracted significant research attention, with approaches spanning various architectural paradigms and task formulations. To provide a comprehensive overview of this rapidly evolving field, we organize prior work along two primary dimensions: 1) task type, distinguishing between layout generation (synthesizing complete layouts from scratch) and layout completion (predicting missing elements in partial layouts); and 2) architectural approach, encompassing GANs [10], VAEs [11], diffusion models [12], and transformer-based methods [13]. Within transformer-based approaches, we further distinguish between encoder-only, decoder-only, and encoder-decoder variants [14], as these architectural choices significantly impact model capabilities and computational efficiency.

A. Layout Generation Approaches

Layout generation has been addressed through various architectural approaches. GAN-based methods like Akin [15] and LayoutGAN [16] utilize adversarial training to synthesize realistic UI wireframes and graphic layouts by modeling geometric relations of 2D elements.

VAE-based approaches, including LayoutVAE [17], LayoutVQ-VAE [18], and VTN [19], offer robust frameworks for generating diverse layouts by learning latent representations and capturing high-level relationships between layout elements.

Recent advancements in layout generation have seen the emergence of diffusion models as a powerful approach for synthesizing structured layouts with high quality and diversity through discrete denoising processes. Notable examples include LayoutDM [20], [22], LayoutDiffusion [21], and layout decoupled diffusion [23].

Transformer-based models have emerged as a powerful approach for layout generation. Gupta et al. [5] introduced LayoutTransformer, a framework that uses self-attention mechanisms to model contextual relationships between layout elements across various domains, enabling the generation of novel scene layouts from an empty set or an initial seed of primitives.

While these layout generation methods have demonstrated impressive capabilities in synthesizing full-screen UI designs, they primarily address the creation of complete layouts from scratch. However, in many practical design scenarios, designers work with partially specified interfaces that require completing existing structures rather than generating entirely new ones.

B. Layout Completion Approaches

A smaller but growing body of work targets layout completion, which is more aligned with iterative design workflows. Li et al. [6] proposed a model for the auto-completion of user interfaces, focusing on assisting the design process by predicting the remaining UI elements in a partial layout. This approach utilizes Transformer-based tree decoders—Pointer and Recursive Transformer—representing a decoder-only architecture to handle the hierarchical structures and 2-dimensional placements of UI elements. Experiments on a public dataset show the effectiveness of these models, which significantly ease the effort of UI designers and developers. The proposed metrics for measuring tree prediction accuracy further contribute to deep learning research in layout prediction.

LayoutFormer++ [24] provides a unified framework for both generation and completion by serializing user constraints and employing a transformer encoder–decoder model. It introduces a constraint serialization scheme to represent various user constraints as token sequences in a predefined format and applies a decoding space restriction strategy to prune predictions that violate user constraints or yield low-quality layouts. This ensures that the generated results adhere to user requirements without sacrificing visual quality.

In contrast to these decoder- and encoder–decoder-based approaches, our work explores an encoder-only transformer formulation for the layout completion task. This design emphasizes parallel processing and self-supervised masked modeling to infer missing elements efficiently. Further details of the proposed method are described in Section III.

While LayoutFormer++ introduces constraint serialization and decoding space restriction to enforce validity during autoregressive generation, LayCoder adopts a fundamentally different approach. Instead of restricting the decoding space step-by-step, our encoder-only formulation relies on global contextual inference through masked modeling. This removes the need for sequential constraint enforcement but places greater emphasis on learning structural regularities directly from data.

Similarly, tree-decoder approaches explicitly preserve hierarchical structure during generation, whereas our flat tokenization strategy prioritizes modeling flexibility and parallel processing. This design choice trades explicit hierarchical inductive bias for scalability and simpler inference, allowing LayCoder to complete layouts in a single forward pass.

III. Proposed Method

Our proposed method section consists of three main components: 1) the definition of key concepts that characterize UI elements, 2) the tokenization strategy that converts structured UI data into a sequential format, and 3) the masked language modeling approach used to train the transformer.

An overview of the system architecture is provided in Fig. 1.

A. Key Concepts

A user interface (UI) consists of a hierarchy of elements, such as buttons, text boxes, images, and containers. These elements are organized as a tree, where some elements may contain others. We classify UI elements into two categories:

- Leaf elements: are elements that do not contain any child elements. Examples include buttons, text inputs, icons, and images.
- Non-leaf elements: are elements that contain one or more child elements. Examples include containers, lists, modals, and cards.

Each element, whether a leaf or non-leaf, is characterized by the following attributes:

- Spatial Coordinates (Bounds): Each element is represented by four values: x_1, y_1, x_2, y_2, denoting the top-left and bottom-right corners of its bounding box on an original screen with a resolution of 2560×1440.
- Element Label (Component Label): A categorical identifier that represents the type of the UI component, such as Text, Image, Container, etc.

In addition, each UI layout is associated with a global attribute:

Layout Category: A label indicating the overall purpose or category of the UI screen (e.g., Login, Registration, Catalog).

These attributes form the foundation for representing UI elements in our model and are critical to both the dataset construction and the tokenization process described in the following sub-sections.

B. Tokenization

We first construct a fixed vocabulary consisting of discrete token values used to encode spatial and semantic attributes of UI elements. For spatial attributes, we discretize normalized coordinate values in the range of 0 to 640, resulting in 641 unique tokens. For semantic attributes, we include 25 tokens representing the Component Labels (e.g., Text, Image), and seven tokens corresponding to distinct Layout Categories identified from the Akin framework [15]: Login, Splash, Registration, Catalog, Product page, Cart, and No-category. Additionally, we define two special tokens—one for separating between elements and one for masking elements during training. This results in a total vocabulary size of 675 tokens.

Spatial discretization to the range [0, 640] provides a consistent vocabulary but introduces bounded quantization error. The maximum absolute discretization error per coordinate is 0.5 units in the normalized scale, corresponding to less than 0.08% of the full spatial extent. While empirically sufficient for preserving layout structure, this discretization limits sub-pixel precision and should be interpreted as an approximation of continuous geometry.

Using this predefined vocabulary, we apply our Layout Tokenizer to discretize the numerical attributes and categorical labels into tokens. Each numerical attribute (e.g., centerX or width) is mapped to a corresponding token ID selected from the constructed vocabulary.

Each UI element is represented by five tokens: four for the spatial attributes (*centerX*, *centerY*, *width*, *height*) and one for the *Component Label*. Each UI layout is then represented as a sequence of tokens, referred to as the *Layout Token Sequence*. As shown in Fig. 2, this sequence starts with the *Layout Category* token, followed by a separator token, and then the *centerX*, *centerY*, *width*, *height*, and *Component Label* tokens for each UI element. Tokens from distinct elements are separated using the special separator token. Consequently, the total length of the sequence is *6N+2*, where N denotes the number of elements in the UI.

A practical aspect of this serialization is the ordering of UI elements. In our implementation, elements follow the order obtained from traversing the hierarchical RICO JSON tree. Although the rendered layout itself is permutation-invariant, Transformer encoders process sequences and are therefore sensitive to element order. We use this traversal order as a deterministic scheme to ensure reproducibility.

Importantly, this representation allows LayCoder to perform masked completion at the level of structured UI elements, where each missing component corresponds to multiple correlated spatial and semantic attributes. This element-level grouping distinguishes the task from a naive application of token-level masked language modeling, as reconstruction operates over structured components rather than independent tokens.

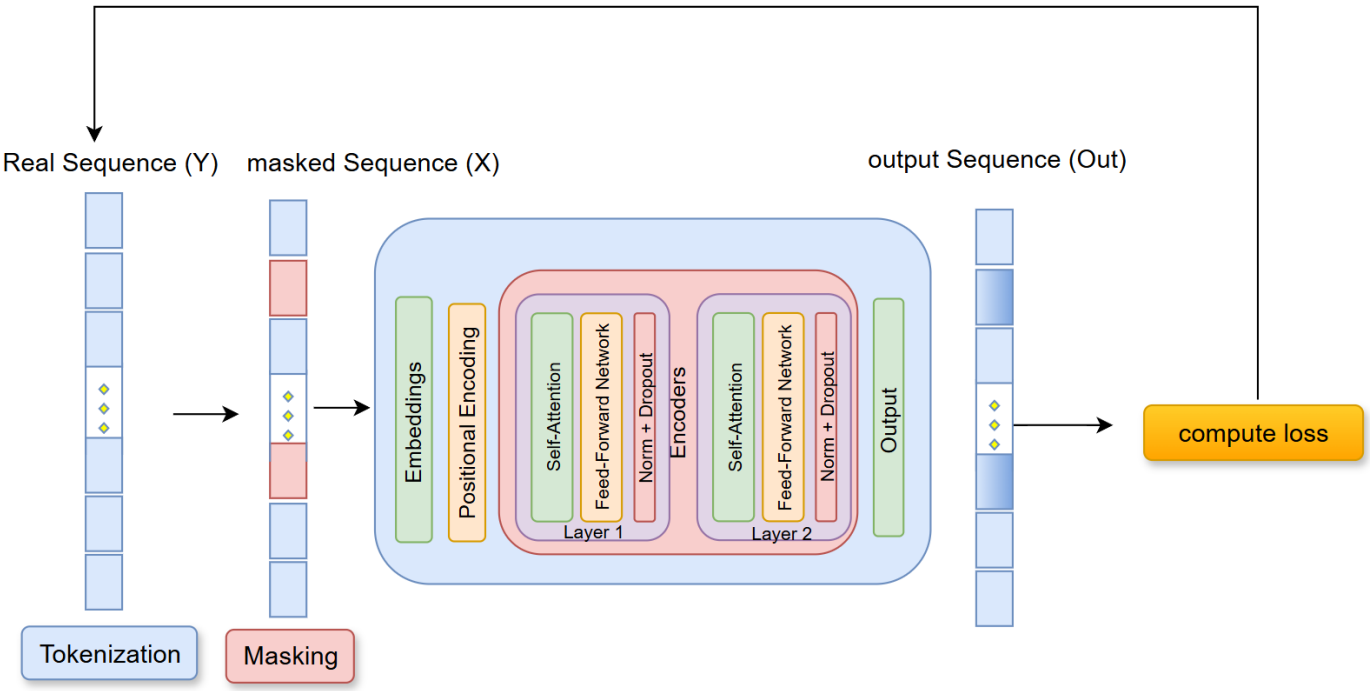

Fig. 1. System architecture: Detailed illustration of the proposed method, including layout tokenization, the masking mechanism, and the two-layer Transformer encoder used for UI layout completion. The architecture highlights how the input token sequence is processed through the encoder to predict the masked UI elements.

This layout-specific tokenization transforms structured UI layouts into flat token sequences that can be processed by the Transformer architecture and can be applied to other datasets with similar structural representations.

C. Encoder Architecture and Masked Training

The architecture follows a standard Transformer encoder stack. Each layer consists of multi-head self-attention, residual connections with layer normalization, and a position-wise feed-forward network. Self-attention enables each token to attend to all other tokens in the layout sequence, capturing long-range spatial and semantic dependencies across UI elements in a single forward pass.

Our approach utilizes an encoder-only transformer architecture, which consists of 2 stacked Transformer layers, each with a hidden size of 512. Within each layer, multi-head self-attention is implemented using 8 attention heads, and the feedforward sublayer has an intermediate dimension of 1024. Layer normalization is applied before each sublayer, and a dropout rate of 0.5 is used within each encoder block to mitigate overfitting. Input tokens are embedded into 512-dimensional vectors using a trainable embedding layer, and positional encodings are added via sinusoidal functions followed by dropout (p = 0.1). The final output is passed through a linear projection layer that maps from the hidden dimension back to the vocabulary size.

We adopt an encoder-only architecture rather than an encoder–decoder model for two reasons. First, masked modeling allows parallel reconstruction of missing elements without autoregressive decoding, reducing inference latency. Second, completion in our setting operates over a partially observed structured sequence, making bidirectional context modeling more natural than left-to-right generation. This design choice favors global contextual reasoning over sequential constraint enforcement.

We incorporate masked language modeling as a core technique of our training objective. It is applied by randomly selecting certain tokens within the layout token sequence and replacing them with our special mask token. During training, this process helps the model learn contextual dependencies between UI elements and their attributes.

Specifically, in the training phase, a random percentage k of tokens from each layout sequence is masked, where $k \in [5\%, 40\%]$. The masked tokens include UI component labels and spatial attributes, while category tokens and separator tokens are excluded to preserve structural consistency. The exact value of k is resampled at the beginning of each epoch. This strategy introduces variability and prevents the model from overfitting to a fixed masking pattern. An example of this token-level masking is illustrated in Fig. 3.

However, during inference, we adopt a masking strategy aligned with LayoutFormer++ [24]. Instead of masking individual tokens independently, we mask complete UI elements—i.e., an element and all of its associated attributes (*Bounds* and *Component Labels*). A fixed masking ratio is applied at the element level rather than the token level. For instance, given a UI layout with 5 elements and a masking ratio of 40%, we mask 2 entire elements, including all tokens representing their attributes. This approach is illustrated in Fig. 4.

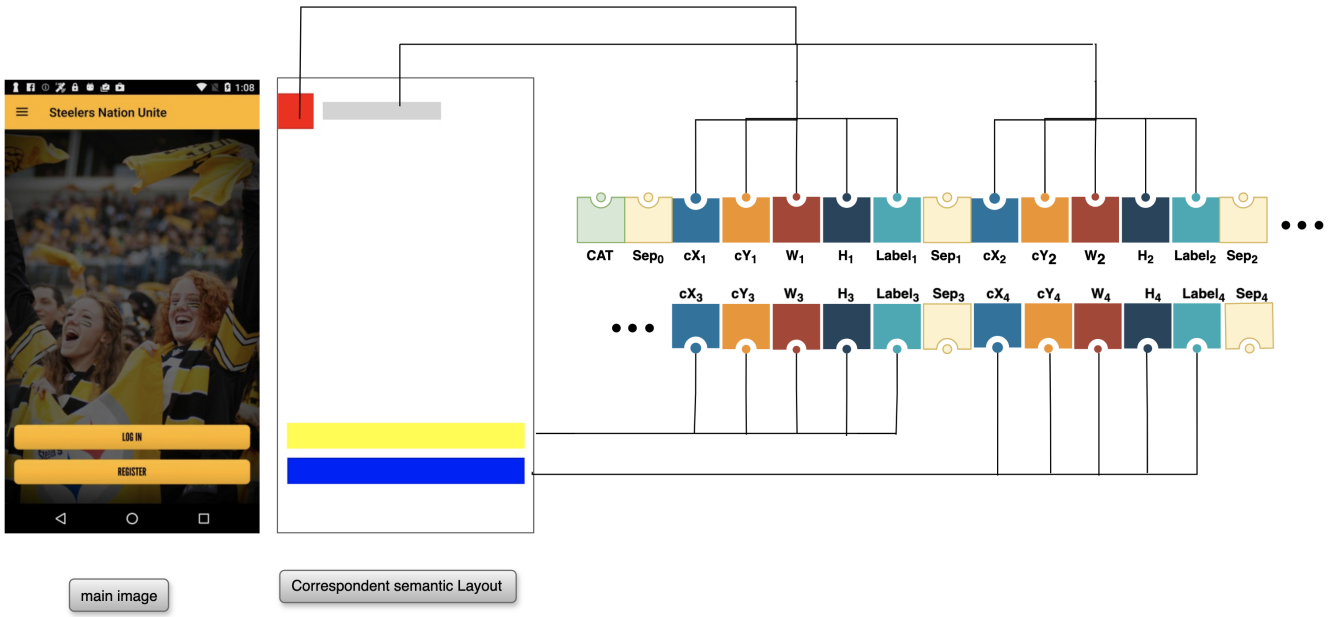

Fig. 2. Overview of the tokenization process using UI as an example with four elements. The Layout Token Sequence begins with the layout category token (CAT), followed by a separator token (Sep). For each UI element, five tokens are appended to the sequence: CenterX (cx), CenterY (cy), width (W), height (H), and the component label (Label). Although the illustration shows four elements, the same procedure applies to layouts with any number of elements.

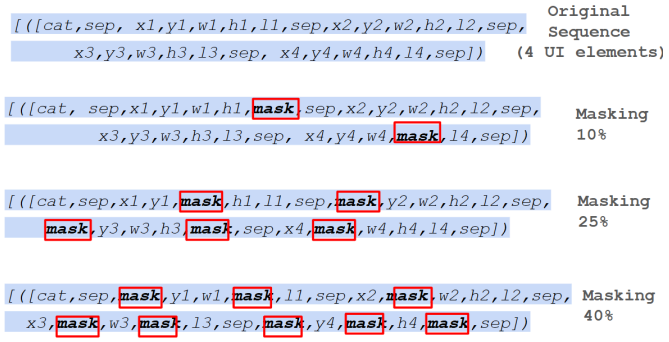

Fig. 3. The masking procedure is applied randomly to each sample in every epoch using a special mask token. The layout category (CAT) and separator (Sep) tokens are never masked; only element attributes are subject to masking. For element i, x_i, y_i, w_i, h_i, and l_i denote centerX, centerY, width, height, and component label, respectively.

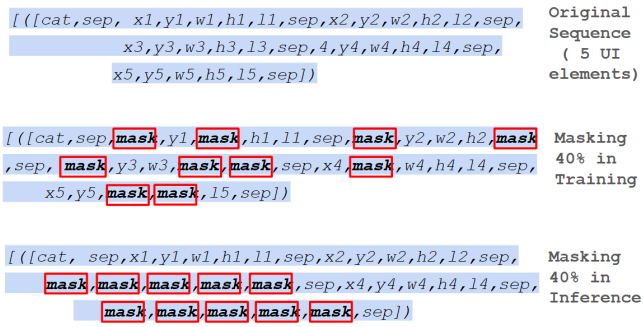

Fig. 4. Comparison of masking strategies between training and inference. During training, masking is applied at the token level, where individual coordinate or label tokens may be masked. During inference, masking is applied at the element level, where all tokens corresponding to a UI element (its coordinates and label) are masked together.

The reason we adopted a different masking strategy during training, as compared to the inference stage, is to learn fine-grained relationships between UI elements, while ensuring that the inference masking setting remains aligned with downstream layout completion scenarios.

The total number of trainable parameters in our configuration is approximately 4.55 million. For a sequence length T, the computational complexity of each encoder layer is $O(T^2)$ due to self-attention. This remains manageable in our setting since layouts are restricted to at most 20 elements (i.e., $T \leq 6N + 2$), enabling efficient training and inference on a single GPU.

IV. Dataset

In this research, we use the RICO dataset [25], a large-scale collection of over 66,000 UI screen images and more than 3 million individual UI elements, collected from a wide variety of Android mobile applications. RICO is one of the most comprehensive and diverse benchmarks for mobile user interfaces, providing rich resources for UI layout generation and completion design tasks.

Each UI screen in the dataset is represented in three formats: 1) a high-resolution screenshot, 2) a corresponding *semantic layout image* that visualizes the structure of the UI using colored bounding boxes (where each color represents a different UI element type, and the size of each box matches

Fig. 5. UI labels color palette: All UI component labels and their corresponding RGB color assignments.

the element's actual size on the original screen), and 3) a JSON file containing detailed metadata. This metadata includes attributes, bounding box coordinates, Android class names, and the hierarchical relationships among elements.

To ensure consistency across all visualizations, the *semantic layout images* use a predefined *UI color palette*, in which each component label is assigned a unique RGB color. Fig. 5 presents the complete set of component labels used in this research together with their corresponding color codes.

A. Re-organizing the RICO Dataset

Following prior work [24], [5], [15], we do not use the RICO dataset in its raw form. Instead, we construct subsets specifically designed to meet the requirements of our proposed method. This involves re-organizing the dataset structure and preprocessing the raw UI layout data accordingly.

As previously described, each UI screen in RICO is accompanied by a JSON file that defines its layout as a hierarchical tree of UI elements. By recursively traversing this tree, we determine the depth of each element and classify them into two categories: leaf elements (elements with no child nodes) and non-leaf elements (elements with one or more children), as described in Section III-A. These distinctions play a central role in how we construct dataset variants for our experimental setups.

To ensure fair comparison with prior work, we build upon the same processed LayoutFormer++ [24] dataset and derive two subset variants, each tailored to a specific experimental objective.

- DS1: This subset corresponds to the version used by LayoutFormer++ during its preprocessing pipeline, in which all UI layouts with more than 20 elements were removed. The result is a curated dataset where each instance contains at most 20 elements. *DS1* consists of 31,694 training samples (221,858 elements), 3,729 test samples, and 1,865 validation samples.
- DS2: This subset is a refined version of *DS1*. Two additional filters are applied: 1) only *leaf elements* are retained, while all *non-leaf elements* are discarded, and 2) any layouts containing spatially overlapping elements are excluded to ensure clean, non-overlapping compositions.

Since our experiments are built upon the same processed subset and data-splitting protocol introduced in LayoutFormer++, the train/test separation follows the established benchmark setting. Layouts are split at the screen-instance level such that no identical UI structures appear across DS1 training and DS2 testing.

However, because RICO contains many visually similar screens within the same application, this protocol does not fully eliminate potential app-level similarity. Consequently, the reported results should be interpreted as generalization across screen instances under the standard benchmark split, rather than strict generalization across entirely unseen applications.

The additional filtering applied in DS2 is motivated by reducing what we refer to as layout noise, namely structural artifacts that may obscure clear spatial patterns. Such noise may arise both from inherent layout complexity and from dataset construction artifacts. In our setting, this includes overlapping elements and non-leaf elements, which can introduce redundancy or hierarchical complexity that is not essential for the layout completion task. By filtering out these elements, we simplify the input representation and promote more consistent spatial modeling and better generalization.

At the same time, this simplification comes with an important modeling trade-off, since UI layouts are inherently hierarchical in nature. However, removing non-leaf elements also eliminates explicit parent–child relationships that encode containment and compositional structure within the UI hierarchy. As a consequence, the model no longer directly observes grouping patterns, nesting depth, or structural dependencies that are central to UI semantics. The DS2 configuration should therefore be interpreted as a structural ablation that isolates spatial reasoning from hierarchical reasoning, rather than as a universally superior representation.

B. Preprocessing

In the preprocessing stage, we normalize the *Bounds* to fit within a 0 to 640 scale, corresponding to the maximum height of the *semantic layout images*. Subsequently, transformations are applied to convert these Bounds into *centerX*, *centerY*, *width*, and *height*, with the former two indicating the center of the element, and the latter two indicating the width and height of the UI element's bounding box.

By transforming the bounds coordinates and normalizing their values, we ensure that the positional information of UI elements is accurately represented within a consistent scale.

The transformation to center-based coordinates allows for more intuitive spatial representations, facilitating the model's ability to understand the layout and structure of the UI elements [27]. The vocabulary described in Section III-B is also constructed as part of this preprocessing step.

V. Evaluation Metrics

In our study, we employed the same evaluation methods used by previous researchers. While our experiments focus on layout completion and some prior studies focus on layout generation, we use the same evaluation metrics in both cases, as they are equally applicable to assessing spatial and structural layout quality. The metrics used in this study are explained as follows:

Coverage Metric is calculated as the proportion of the UI layout area that is occupied by layout elements. In other words, it is the area covered by elements divided by the total area of the canvas (UI layout). Higher scores indicate better coverage, reflecting layouts that are more fully and effectively populated with UI components.

IoU measures the degree to which layout elements overlap within a completed layout. Specifically, it is the average Intersection over Union between each pair of elements (bounding boxes) within the same layout. Mathematically, IoU is defined as the area of Intersection divided by the area of the Union between two bounding boxes, providing a normalized measure that accounts for both the intersecting and non-overlapping regions. The computation of IoU between two elements *A* and *B* is shown in the following equation:

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

where, $|A \cap B|$ denotes the area of their intersection, and $|A \cup B|$ denotes the area of their union. Lower scores indicate better IoU, ensuring that no overlap is occurring.

Max IoU provides insight into the maximum overlap any element in a layout has with other elements. It is obtained by first computing the pairwise IoU between each element within a UI, then, for each element, the maximum IoU value is retrieved, and finally, those values are summed and averaged. Similar to IoU, lower Max IoU values are preferable.

Alignment is a metric originally introduced in the Attribute-conditioned Layout GAN paper [26] and later adopted in LayoutFormer++ [24] to evaluate the spatial alignment of UI elements. For each layout element, we compute six reference points: the left, center, and right positions along the horizontal axis, and the top, center, and bottom positions along the vertical axis. Pairwise distances are then calculated between each of these points and the corresponding points of all other elements in the layout. For each element, the minimum of these distances is taken as its alignment error with respect to the rest of the layout.

Finally, the alignment scores are summed and averaged, producing a single scalar value per layout. Lower scores indicate better alignment, reflecting a more structured and visually coherent layout.

These metrics provide a comprehensive evaluation framework by measuring element distribution, spatial overlap, and spatial coherence within each layout. Specifically, Coverage reflects the overall distribution and presence of elements across the layout, IoU and Max IoU quantify the degree of element overlap, and Alignment evaluates spatial coherence.

At the same time, these metrics primarily capture geometric and structural properties of the completed layouts. They do not directly measure semantic correctness of element identities. A completion may achieve strong spatial alignment while predicting an incorrect component label. Therefore, the current evaluation emphasizes spatial fidelity rather than full element-wise reconstruction accuracy.

VI. Experiments

In this section, we trained our models on two curated subsets of the RICO dataset, *DS1* and *DS2*, introduced in Section IV. Our models were trained to infer and complete UI layouts with up to 40% of elements randomly masked. More specifically, we systematically varied the masking rate of UI elements from 5% to 40% during training and inference, in order to evaluate the models with different levels of masked information.

Furthermore, model performance is assessed using the evaluation metrics described in the previous Section: Coverage, IoU, Max IoU, and Alignment. The original, unmasked UI layouts serve as ground-truth references, and model performance is evaluated using the defined metrics by comparing the predicted layouts with these references.

A. Model Variants

We trained two variants of our proposed model using the aforementioned datasets:

- LayCoder — a model trained on *DS1*, containing up to 20 elements per UI layout, and including both leaf and non-leaf nodes. The model was trained using the same preprocessing steps as [24].
- LayCoder-NonOver-Leaves — a model trained on *DS2*, a refined version of DS1 containing only leaf nodes and no overlapping elements. This model leverages the simplified structure of the data (leaf-only nodes and non-overlapping elements) to evaluate the impact of dataset noise reduction on model performance.

We aim with the first variant to demonstrate that our proposed model, *LayCoder*, achieves superior performance compared to the baseline, *LayoutFormer++*, across standard layout evaluation metrics.

With *LayCoder-NonOver-Leaves*, we specifically test the hypothesis that reducing layout noise—through the exclusion of overlapping elements and non-leaf nodes—can enhance model performance. Though we evaluate both models on the same test set (the testing subset of *DS2*). *LayCoder* was trained on noisier and more diverse layouts (*DS1*), while *LayCoder-NonOver-Leaves* was trained on a less noisy dataset (*DS2*). We ensure as well that the test set of *DS2* does not overlap with the training data of *DS1*. This setup allows us to examine the importance of curating the data to contain only elements relevant to our task and measure its impact on the model's ability to generalize and accurately infer UI structures.

B. Experimental Setup

All experiments were conducted on an NVIDIA RTX 3090 GPU. During training, random masking between 5% and 40% was applied to the token sequence, as described in the Method section. We adopted cross-entropy loss, computed only over the masked tokens; unmasked tokens, as well as special tokens and layout category tokens, were excluded from the loss calculation. The model predicts each masked token from a predefined vocabulary of 666 discrete tokens, which includes 641 tokens representing discretized bounding box attributes and 25 tokens corresponding to element labels. The model is trained using the Adam optimizer with an initial learning rate of 1e-4. A batch size of 64 is used, and training proceeds for 1000 epochs with early stopping based on validation loss, which is monitored every 100 epochs.

VII. Results and Discussion

A. Quantitative Results

Experimental results are detailed in Table I and Table II. In these tables, the row labeled as *Real Data* indicates the mean values for each evaluation metric computed over the unmasked test subsets of DS1 and DS2, serving as the reference baseline. These values represent the expected metric scores for complete, unmasked layouts. The preceding rows, labeled as *10% masking*, *25% masking*, and *40% masking*, indicate the performance of each model when completing UI designs with 10%, 25%, and 40% of masked elements, respectively.

In Table I, we compare the performance of our first model, the *LayCoder*, with the *LayoutFormer++* [24] on the testing subset of *DS1*, which includes a mix of leaf and non-leaf elements.

As introduced in Section V, the performance of the models is evaluated using the following metrics: Coverage, IoU, Max IoU, and Alignment. Furthermore, the Δ value for each metric denotes the absolute difference between the layouts produced by each model and the reference baseline:

$$\Delta = |\text{Predicted} - \text{Baseline}| \tag{2}$$

Therefore, a smaller Δ indicates that the predicted layouts closely approximate the original data.

In the first row of Table I, when masking 10% of the data, we observe a substantial performance difference, particularly in Coverage. *LayCoder* achieves a score very close to the real data, with $\Delta = 1.76$, whereas *LayoutFormer++* shows a much higher $\Delta = 32.38$, indicating severe under-coverage. For IoU and Max IoU, *LayCoder* also achieves smaller Δ values (0.002 and 0.013, respectively) compared to *LayoutFormer++* (0.022 and 0.065). These results indicate closer alignment with the reference layouts in terms of overlap. In terms of Alignment, *LayCoder* achieves a perfect $\Delta = 0$, meaning that its completions under 10% masking are equivalent to the baseline layouts without masking. This outcome is consistent with the low masking ratio, where fewer elements are removed and reconstruction is less challenging.

As the masking ratio increases to 25%, *LayCoder* continues to outperform *LayoutFormer++* across all metrics. For Coverage, LayCoder's $\Delta = 3.20$ is still much closer to the real data than LayoutFormer++ ($\Delta = 32.17$). Similarly, for IoU ($\Delta = 0.007$ vs. 0.022), Max IoU ($\Delta = 0.037$ vs. 0.064), and Alignment ($\Delta = 0.001$ vs. 0.004), LayCoder remains stronger. Although its scores show a small degradation compared to the 10% case, this is expected as the masking ratio increases. By contrast, LayoutFormer++ shows little change compared to its 10% performance. This relative stability may be attributed to its additive, sequential training strategy, where elements are generated incrementally. As a result, increasing the masking percentage does not substantially alter its completion behavior.

At 40% masking, both models face more challenging conditions, but *LayCoder* still demonstrates stronger generalization. Its Coverage $\Delta = 3.47$ remains far superior to LayoutFormer++ ($\Delta = 32.37$), suggesting that LayCoder better preserves the overall density of elements under high masking. For Alignment, both models are close ($\Delta = 0.003$ vs. 0.002). On IoU and Max IoU, LayCoder records $\Delta = 0.017$ and 0.072, while LayoutFormer++ remains at 0.024 and 0.066. Although LayoutFormer++ appears slightly better in Max IoU at this level, its Coverage remains very poor ($\Delta \approx 32$ across masking levels).

Since Coverage measures the proportion of the layout canvas occupied by elements, such low values suggest that many UI components are missing, leading to incomplete layouts that deviate substantially from the ground truth. This behavior is closely related to the differing completion paradigms of the two models. LayoutFormer++ generates elements autoregressively in a fixed sequential manner without explicit conditioning on the final number of elements to be produced. As a result, it often yields conservative predictions under high-masking settings, leading to systematic under-coverage and relatively stable scores across masking ratios.

In contrast, LayCoder performs masked reconstruction within a fixed-length sequence, where the number of missing elements is implicitly determined by the masked slots. This formulation encourages preservation of element count and overall layout density. Both models were evaluated under identical masking ratios and metric implementations to ensure consistency.

In summary, across all three masking levels (10%, 25%, 40%), *LayCoder* consistently produces results closer to the real data. Its Coverage is significantly better than *LayoutFormer++* in every experiment, with Δ values ranging from 1.76 to 3.47 compared to around 32 for *LayoutFormer++*. In IoU, Max IoU, and Alignment, *LayCoder* also maintains lower or comparable Δ values, showing stronger robustness under increasing masking ratios.

While the performance of *LayoutFormer++* appears stable across masking levels, this stability reflects the limitations of its autoregressive decoder-based architecture. Because it generates layouts sequentially, the model appears less sensitive to the amount of masked content and shows limited adaptability as masking increases. By contrast, *LayCoder*'s encoder-only masked modeling leverages global layout context, enabling it to adapt better to different masking conditions, preserve spatial structure, and maintain performance much closer to the real-data.

The observed results in Table II indicate that in terms of Coverage, *LayCoder* remains consistently closer to the real

TABLE I. COMPARATIVE RESULTS ON DS1 AT DIFFERENT MASKING LEVELS

Masking	Model	Coverage ↑ (Mean, Δ)	Overlap (IoU) ↓ (Mean, Δ)	Max IoU ↓ (Mean, Δ)	Alignment ↓ (Mean, Δ)
10%	LayCoder	**74.88 ($\Delta = 1.76$)**	0.055 (**$\Delta = 0.002$**)	0.190 (**$\Delta = 0.013$**)	**0.014 ($\Delta = 0.000$)**
	LayoutFormer++	**44.26** ($\Delta = 32.38$)	**0.031** ($\Delta = 0.022$)	**0.112** ($\Delta = 0.065$)	**0.019** ($\Delta = 0.005$)
25%	LayCoder	**73.44 ($\Delta = 3.20$)**	0.060 (**$\Delta = 0.007$**)	0.214 (**$\Delta = 0.037$**)	**0.013 ($\Delta = 0.001$)**
	LayoutFormer++	**44.47** ($\Delta = 32.17$)	**0.031** ($\Delta = 0.022$)	**0.113** ($\Delta = 0.064$)	**0.018** ($\Delta = 0.004$)
40%	LayCoder	**73.17 ($\Delta = 3.47$)**	0.070 (**$\Delta = 0.017$**)	**0.249** ($\Delta = 0.072$)	**0.011** ($\Delta = 0.003$)
	LayoutFormer++	**44.27** ($\Delta = 32.37$)	**0.029** ($\Delta = 0.024$)	**0.111 ($\Delta = 0.066$)**	0.016 (**$\Delta = 0.002$**)
Real Data		**76.64**	**0.053**	**0.177**	**0.014**

A comparative evaluation between LayCoder and LayoutFormer++ at different masking levels using the test set from DS1, which includes both leaf and non-leaf UI elements. Real Data at the bottom of the table represents metrics computed on unmasked UI layouts as a ground-truth reference.

TABLE II. COMPARATIVE RESULTS ON DS2 AT DIFFERENT MASKING LEVELS

Masking	Model	Coverage ↑ (Mean, Δ)	IoU ↓ (Mean = Δ)	Max IoU ↓ (Mean = Δ)	Alignment ↓ (Mean, Δ)
10%	LayCoder-NonOver-Leaves	41.21 (**$\Delta = 0.49$**)	**0.002**	**0.024**	0.0183 (**$\Delta = 0.0004$**)
	LayCoder	**42.30** ($\Delta = 0.60$)	0.003	0.033	**0.0179** ($\Delta = 0.0008$)
25%	LayCoder-NonOver-Leaves	38.80 (**$\Delta = 2.90$**)	**0.008**	**0.058**	0.0204 (**$\Delta = 0.0017$**)
	LayCoder	**41.43 ($\Delta = 0.27$)**	0.015	0.089	**0.0162** ($\Delta = 0.0025$)
40%	LayCoder-NonOver-Leaves	37.00 (**$\Delta = 4.70$**)	**0.014**	**0.092**	0.0198 (**$\Delta = 0.0011$**)
	LayCoder	**41.89 ($\Delta = 0.19$)**	0.032	0.144	**0.0137** ($\Delta = 0.0050$)
Real Data		**41.70**	**0.000**	**0.000**	**0.0187**

Evaluation of LayCoder and LayCoder-NonOver-Leaves under different masking levels, using test samples from DS2, which contains only non-overlapping leaf elements. Note that the ground-truth reference scores (Real Data) for Intersection over Union (IoU) and maximum IoU are zero; thus, the reported mean values are equal to their corresponding Δ values.

data across all masking levels (Δ = 0.60, 0.27, and 0.19 at 10%, 25%, and 40%, respectively), whereas *LayCoder-NonOver-Leaves* shows larger deviations (Δ = 0.49, 2.90, and 4.70). This confirms that the leaves-only, non-overlapping training setup sacrifices Coverage, leading to layouts that omit some elements compared to the ground truth.

In contrast, for the spatial consistency metrics (IoU, Max IoU, and Alignment), *LayCoder-NonOver-Leaves* generally outperforms *LayCoder*. For instance, at 25% masking, it achieves lower Δ IoU (0.008 vs. 0.015) and Δ Max IoU (0.058 vs. 0.089), indicating reduced element overlap and stronger alignment with ground-truth structures. At 40% masking, *LayCoder* achieves a lower mean Alignment score (0.0137 vs. 0.0198). However, *LayCoder-NonOver-Leaves* reports lower Δ values for IoU, Max IoU, and Alignment, indicating stronger spatial consistency with the ground-truth.

Overall, these findings suggest that training on clean, non-overlapping leaves improves spatial consistency in terms of overlaps and relative positioning, but at the cost of reduced Coverage. Thus, the results highlight a clear trade-off: while curated datasets can strengthen spatial precision, they may also lead to incomplete reconstructions that fail to capture all layout elements present in the original designs.

B. Qualitative Analysis

To complement the quantitative results, we also conducted a qualitative analysis to examine the behavior of our proposed models under different masking conditions. For readability, the layouts are displayed using the same color-coded scheme introduced in Section IV, with component labels distinguished by the *UI color palette* (see Fig. 5).

Fig. 6 illustrates the predictions of *LayCoder* on one sample from the DS1 test set under three masking ratios: 10%, 25%, and 40%. For each ratio, the masked input and the completed layout are shown, while the corresponding ground-truth layout (labeled as "Real UI layout" in the figure) is displayed at the bottom. In many cases, the model successfully reconstructs missing elements and maintains a coherent spatial structure. The completions are not always identical to the ground truth, and in some cases the model generates alternative components or placements. While these differences can still yield plausible layouts from a design perspective, we emphasize that this observation is based on illustrative examples rather than systematic evaluation.

For further assessment, Fig. 7 presents results on three different UI layouts from DS1, each with a different masking ratio (10%, 25%, and 40%). Alongside reasonable completions, the figure highlights typical failure modes such as overlapping components, duplicated elements in the same region, or missing items. These examples demonstrate that although *LayCoder* can often predict plausible completions, it may also produce undesirable layouts under challenging conditions, especially at higher masking levels.

In Fig. 8, we evaluate both *LayCoder* and *LayCoder-NonOver-Leaves* on samples from DS2, where only leaf elements are retained and overlapping layouts are excluded. Using a 40% masking ratio, we compare the completed layouts of both models against the ground truth. The figure shows that *LayCoder-NonOver-Leaves* often avoids generating overlapping elements and produces cleaner structures, which is consistent with the hypothesis that reducing layout noise during training improves spatial consistency. However, limitations remain: in some cases, the outputs omit required elements or contain misaligned components, reflecting the cost

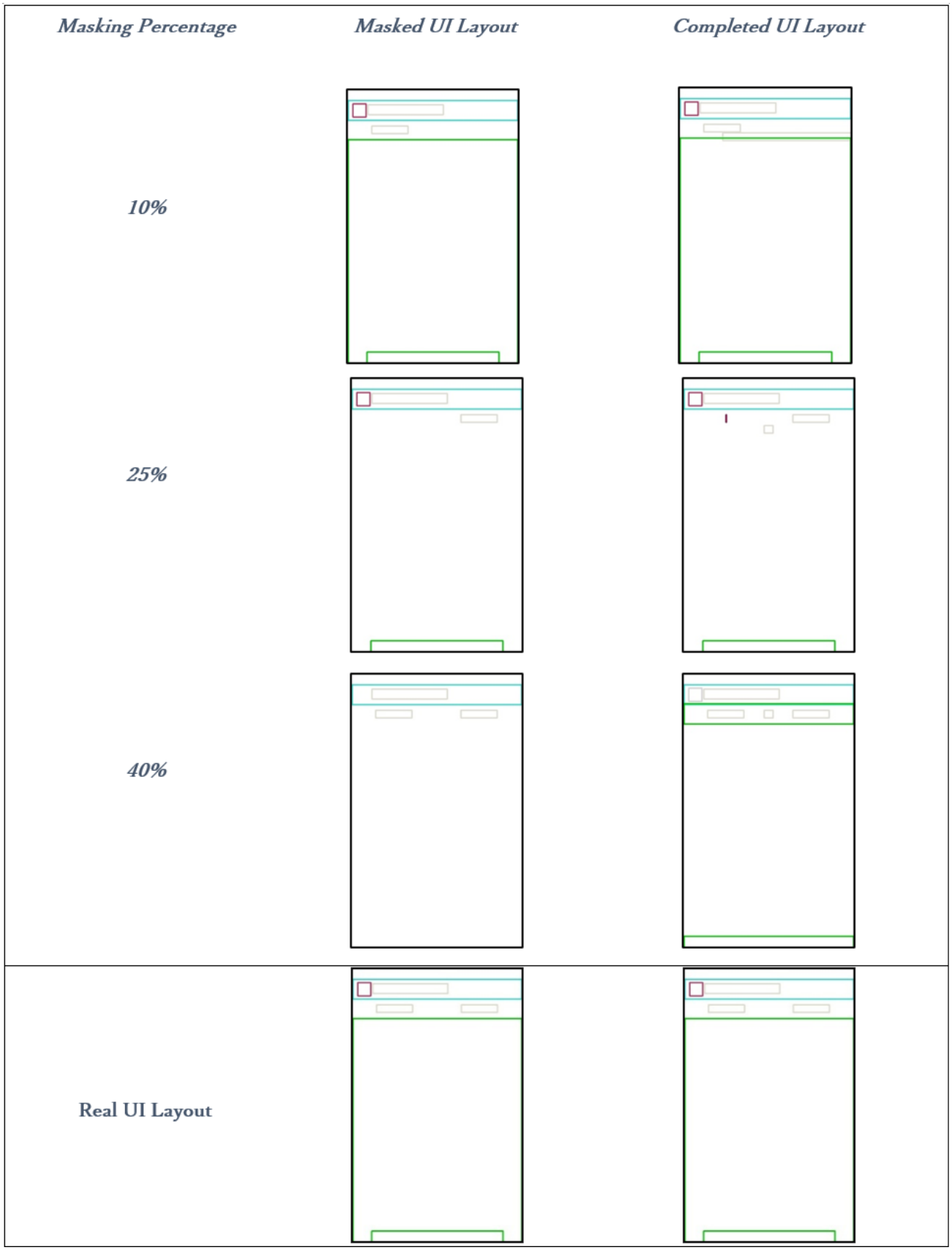

Fig. 6. Qualitative results of LayCoder on a DS1 sample under 10%, 25%, and 40% masking. Each row shows a masked input with the corresponding masking ratio and the resulting completion by LayCoder. The last row shows the ground-truth (real UI) layout.

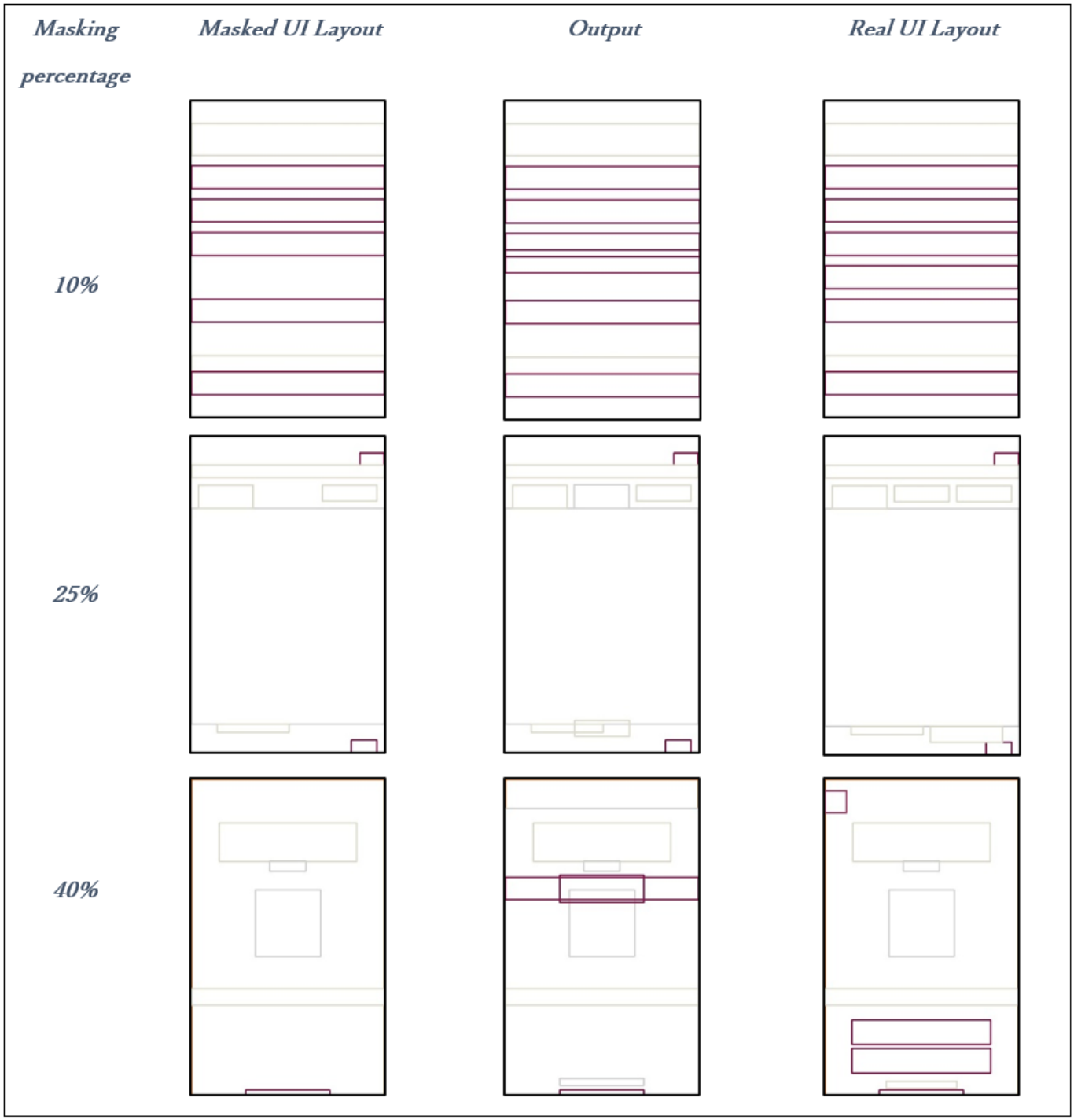

Fig. 7. Some failure cases from DS1 under different masking ratios. Examples highlight overlaps, duplicated elements, and missing components in *LayCoder* outputs.

of training on a more constrained dataset. Thus, while dataset curation enhances certain aspects of spatial accuracy, it does not guarantee uniformly superior results.

Overall, the qualitative evaluation highlights both the strengths and weaknesses of the proposed models. *LayCoder* demonstrates the ability to infer missing elements and produce plausible layouts across different masking levels, while *LayCoder-NonOver-Leaves* benefits from noise reduction by yielding cleaner structures. At the same time, both models exhibit failure cases that reveal important directions for further improvement.

Fig. 8. Comparison between *LayCoder* and *LayCoder-NonOver-Leaves* on DS2 with 40% masking. For each layout example, the masked input is shown alongside the completed outputs of both models and the corresponding ground-truth (real) UI layout.

VIII. Conclusion

In this study, we introduced LayCoder, an encoder-only Transformer formulated as a masked completion model for UI layout reconstruction. Our experiments show that LayCoder achieves competitive performance compared to LayoutFormer++, delivering improved Coverage and comparable or improved IoU and Alignment across the evaluated masking levels.

The results indicate that encoder-only transformer architectures combined with masked modeling are effective for layout completion under the evaluated settings. This formulation enables the model to leverage global context and learn structural dependencies without requiring explicit element-level annotations.

To further investigate the role of training data, we evaluated a noise-reduced subset of the dataset that excluded overlapping elements and non-leaf nodes. The results show that this curation can improve spatial consistency in terms of overlap and relative positioning, but at the cost of reduced Coverage. These findings suggest that dataset filtering shifts the balance of performance: strengthening certain structural aspects while limiting completeness, rather than uniformly enhancing predictions.

Despite these results, the current formulation relies on flat token representations and relatively shallow architectures, and does not explicitly model hierarchical relationships or complex structural constraints. Therefore, the findings should be interpreted as demonstrating the viability of masked encoder-only completion within the evaluated regime, rather than establishing scalability to deeper hierarchical modeling.

Failure cases reveal issues such as overlaps, duplications, or missing elements under challenging masking conditions, reflecting a structural trade-off between element completeness and precise spatial alignment. This trade-off suggests that incorporating additional structural constraints or relational representations may further improve the balance between layout completeness and spatial coherence in future work.

Acknowledgment

The authors would like to thank the creators of the *Rico dataset* and the authors of *LayoutFormer++* for making their data and code publicly available, which facilitated this research.

Data Availability

The datasets used in this study were derived from the publicly available *LayoutFormer++* dataset, which is based on the Rico dataset and available at https://github.com/microsoft/LayoutGeneration/tree/main/LayoutFormer++. We further processed this dataset to create two subsets, **DS1** and **DS2**, provided as `.pkl` files. All processed data and related preprocessing scripts are publicly available in our GitHub repository at https://github.com/salsik/LayCoder-Data under the MIT license.

Code Availability

The source code implementing the LayCoder model and training procedures is available at the project repository: https://github.com/salsik/cssl.

References

[1] Majumder, A. S. 2025. The Influence of UX Design on User Retention and Conversion Rates in Mobile Apps. ArXiv.

[2] Troiano, Luigi, and Birtolo, Cosimo. 2014. Genetic algorithms supporting generative design of user interfaces: Examples. Information Sciences, 259 (February 2014), 433-451. https://doi.org/10.1016/j.ins.2012.01.006

[3] Bleichner, Andreas, and Nils Hermansson. "Investigating the usefulness of a generative AI when designing user interfaces." (2023).

[4] Bandi, Ajay, Pydi Venkata Satya Ramesh Adapa, and Yudu Eswar Vinay Pratap Kumar Kuchi. "The power of generative AI: A review of requirements, models, input–output formats, evaluation metrics, and challenges." Future Internet 15.8 (2023): 260.

[5] Gupta, Kamal, et al. "LayoutTransformer: Layout generation and completion with self-attention." Proceedings of the IEEE/CVF International Conference on Computer Vision. 2021.

[6] Li, Yang, et al. "Auto completion of user interface layout design using transformer-based tree decoders." arXiv preprint arXiv:2001.05308 (2020).

[7] Vaswani, A., et al. (2017). "Attention is All You Need." In Advances in Neural Information Processing Systems (NeurIPS).

[8] Devlin, Jacob. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).

[9] Salama I, Mormille LH, Atsumi M. User Interface Design using Masked Language Modeling in a Transformer Encoder-based Model. 2024. doi:10.11517/pjsai.JSAI2024.0_3Q5IS2b05.

[10] Goodfellow, Ian, et al. "Generative adversarial networks." Communications of the ACM 63.11 (2020): 139-144.

[11] Kingma, D. P., & Welling, M. (2014). "Auto-Encoding Variational Bayes." arXiv:1312.6114.

[12] Yang, L., Zhang, Z., Song, Y., Hong, S., Xu, R., Zhao, Y., Zhang, W., Cui, B., & Yang, M.-H. (2022). "Diffusion Models: A Comprehensive Survey of Methods and Applications." arXiv:2209.00796.

[13] Khan, S., Naseer, M., Hayat, M., Zamir, S. W., Khan, F. S., & Shah, M. (2022). "A Comprehensive Survey of Recent Transformers in Image, Video and Other Domains." ACM Computing Surveys (CSUR).

[14] Compare Encoder-Decoder, Encoder-Only, and Decoder-Only Architectures for Text Generation on Low-Resource Datasets. (2021). In *Advances on Broad-Band Wireless Computing, Communication and Applications* (BWCCA).

[15] Gajjar, Nishit, et al. "Akin: Generating ui wireframes from ui design patterns using deep learning." 26th International Conference on Intelligent User Interfaces-Companion. 2021.

[16] Li, Jianan, et al. "Layoutgan: Generating graphic layouts with wireframe discriminators." arXiv preprint arXiv:1901.06767 (2019).

[17] Jyothi, Akash Abdu, et al. "Layoutvae: Stochastic scene layout generation from a label set." Proceedings of the IEEE/CVF International Conference on Computer Vision. 2019.

[18] Jing, Qianzhi, et al. "Layout generation for various scenarios in mobile shopping applications." Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems. 2023.

[19] Arroyo, Diego Martin, Janis Postels, and Federico Tombari. "Variational transformer networks for layout generation." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2021.

[20] Inoue, Naoto, et al. "Layoutdm: Discrete diffusion model for controllable layout generation." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2023

[21] Zhang, Junyi, et al. "Layoutdiffusion: Improving graphic layout generation by discrete diffusion probabilistic models." Proceedings of the IEEE/CVF International Conference on Computer Vision. 2023.

[22] Chai, Shang, Liansheng Zhuang, and Fengying Yan. "Layoutdm: Transformer-based diffusion model for layout generation." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2023.

[23] Hui, Mude, et al. "Unifying layout generation with a decoupled diffusion model." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2023.

[24] Jiang, Zhaoyun, et al. "LayoutFormer++: Conditional Graphic Layout Generation via Constraint Serialization and Decoding Space Restriction." arXiv preprint arXiv:2208.08037 (2022).

[25] Deka, Biplab, et al. "Rico: A mobile app dataset for building data-driven design applications." Proceedings of the 30th annual ACM symposium on user interface software and technology. 2017.

[26] Lee, Hsin-Ying, et al. "Neural design network: Graphic layout generation with constraints." Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part III 16. Springer International Publishing, 2020.

[27] Bai, Zihan, et al. "Layout representation learning with spatial and structural hierarchies." Proceedings of the AAAI Conference on Artificial Intelligence. 2023.