From Screenshots to Hierarchical Code: Android GUI Layout Code Generation via Multi-Agent LLMs

Anonymous ACL submission

Abstract

UI-to-Code systems have achieved strong performance in web interfaces, yet generating structured Android GUI code remains challenging due to layout complexity. We propose a framework that converts Android screenshots into hierarchical code through multi-agent LLMs. The framework begins with GUI component recognition, extracting both local component information and global layout structure. The LLM is then guided to generate code for each component in context, ensuring consistency and modularity. To improve code quality, we introduce a feedback-driven refinement stage that leverages structural similarity metrics for iterative enhancement. We evaluate our approach on subsets of Rico datasets. Results show that our method significantly outperforms Pix2Code, direct prompting, and chain-of-thought prompting strategies. Our findings highlight the effectiveness of layout-aware prompting and structured refinement for accurate Android GUI code generation.

1 Introduction

Recent years have witnessed increasing interest in automatically generating user interface (UI) code from visual designs or screenshots, known as the UI-to-Code task. This automation streamlines development by reducing the manual effort of translating mockups into code. In the web domain, research has progressed rapidly—from early neural models to modern prompt-based approaches. The seminal PIX2CODE model (Beltramelli, 2017) first showed that end-to-end deep learning could generate UI code for web, iOS, and Android platforms from a single screenshot. More recently, Wan et al. (Wan et al., 2024) demonstrated that segmenting a webpage and prompting an LLM on each region improves visual similarity by 14% over holistic prompting. Despite such advances in web UI generation, the Android UI-to-Code problem remains underexplored and presents distinct structural challenges.

Android GUI code generation poses unique challenges due to the structured and deeply nested nature of layout definitions in XML. Unlike HTML, Android layouts rely on containers such as `RelativeLayout` and `ConstraintLayout`, which require precise spatial reasoning and are not directly renderable without an IDE or emulator. This limits the use of visual feedback and image-based losses common in web UI generation. Moreover, paired screenshot–XML data is scarce. While datasets like Rico (Deka et al., 2017a) provide view hierarchies, they lack exact layout annotations. Prior efforts (Chen et al., 2018) constructed large-scale datasets via automated app exploration, but such resources remain limited. These factors—structural complexity and data scarcity—have constrained progress in Android UI-to-code compared to the web domain.

Despite its challenges, early work established the feasibility of data-driven GUI code generation. PIX2CODE (Beltramelli, 2017) framed the task as sequence-to-sequence translation, using a CNN–RNN architecture to convert screenshots into platform-specific code. On Android, UI2CODE (Chen et al., 2018) learned to predict layout hierarchies from design images without handcrafted rules, leveraging a large corpus of screenshot–layout pairs collected via automated app exploration. More recent systems, including both commercial tools and academic prototypes, explore prompting large language models (LLMs) with UI images or descriptions. However, one-shot prompting often leads to missing elements, incorrect hierarchies, or hallucinated structures—issues especially pronounced in Android XML generation. These limitations highlight the need for more structured and guided generation strategies.

A promising direction for UI-to-code generation is to leverage large language model (LLM)

agents that decompose the task into structured subtasks and solve it hierarchically. Unlike single-pass models, LLM agents can iteratively reason over the interface, invoke specialized steps, and refine intermediate outputs. Such multi-step planning improves reliability by reducing hallucinations and error accumulation (Yao et al., 2022). We argue that Android UI generation, with its structural complexity, benefits from this approach. By focusing on one component or region at a time and incorporating feedback at each stage, the agent can maintain global layout coherence while systematically generating XML code. This hierarchical process aligns with the strengths of LLMs—contextual understanding and code generation—while introducing structure to mitigate their weaknesses.

To address these challenges, we propose a layout-aware framework that transforms Android GUI screenshots into native XML code using a single large language model (LLM) guided by structured prompting. The process begins with GUI component recognition, which identifies individual UI elements and captures their spatial relationships. The LLM is then sequentially prompted to generate partial code for each component, conditioned on its local context and global layout. These fragments are composed into a complete XML hierarchy, preserving structural and visual fidelity. To further improve output quality, we introduce a reinforcement-style refinement stage, where feedback is used to iteratively correct omissions and structural errors.

We evaluate our method on subsets of the Rico dataset (Deka et al., 2017a), and show that it significantly outperforms traditional UI-to-code models and prompting-based LLM baselines in both structural accuracy and component-level metrics. Our approach highlights how structured planning and iterative prompting can enhance LLM capabilities in GUI code generation, bridging visual understanding with language-based reasoning for high-fidelity Android layout synthesis.

2 Related Work

2.1 UI-to-Code

Early UI-to-code approaches relied on image processing and template-based heuristics. REMAUI (Nguyen et al., 2015) used OCR and computer vision to extract text, images, and components from app screenshots, reconstructing view hierarchies for Android UI generation. Sketch2Code applied similar techniques to hand-drawn wireframes. While effective for simple designs, these methods struggled with complex layouts due to their reliance on hand-crafted rules and rigid pipelines.

The introduction of deep learning marked a shift to more generalizable models. PIX2CODE (Beltramelli, 2017) pioneered end-to-end UI generation using a CNN encoder and sequence decoder, achieving over 77% accuracy on synthetic datasets across web, iOS, and Android. Chen et al. (Chen et al., 2018) extended this to Android with a dataset of 185k screenshot–code pairs and a CNN–RNN architecture that captured layout hierarchies across varied designs. ReDraw (Moran et al., 2018) combined CNN-based classification with mining software repositories to assemble structurally faithful Android prototypes.

More recently, transformer-based models have treated UI-to-code as a multimodal translation task, mapping GUI images to structured outputs such as HTML or JSON using self-attention (Liu et al., 2021). Meanwhile, large LLMs have been prompted with UI descriptions or screenshots to generate code directly. However, one-shot prompting often fails on deep layouts or dense hierarchies. Si et al. (Si et al., 2024) show that LLM performance deteriorates as the number of UI elements and nesting depth increase, especially in Android settings. These limitations highlight the need for layout-aware and stepwise generation strategies.

2.2 LLM Agents

Recent advances in LLM-based agent frameworks have enabled complex, multi-step reasoning that benefits structured generation tasks. Rather than producing outputs in a single pass, LLM agents interleave reasoning with actions, decomposing tasks, invoking tools, and refining results.

ReAct (Yao et al., 2022) exemplifies this approach by interleaving chain-of-thought reasoning with tool-use actions, allowing models to plan steps and fetch factual information when needed. Toolformer (Schick et al., 2023) further demonstrated that LLMs can self-supervise API calls (e.g., calculators, search engines), enhancing zero-shot performance by offloading sub-tasks to external tools. These systems highlight how action-aware prompting improves reliability and reduces hallucinations.

Beyond tool invocation, agent frameworks like AutoGPT and BabyAGI introduce autonomous planning loops, where an LLM iteratively generates goals, executes actions, and evaluates progress

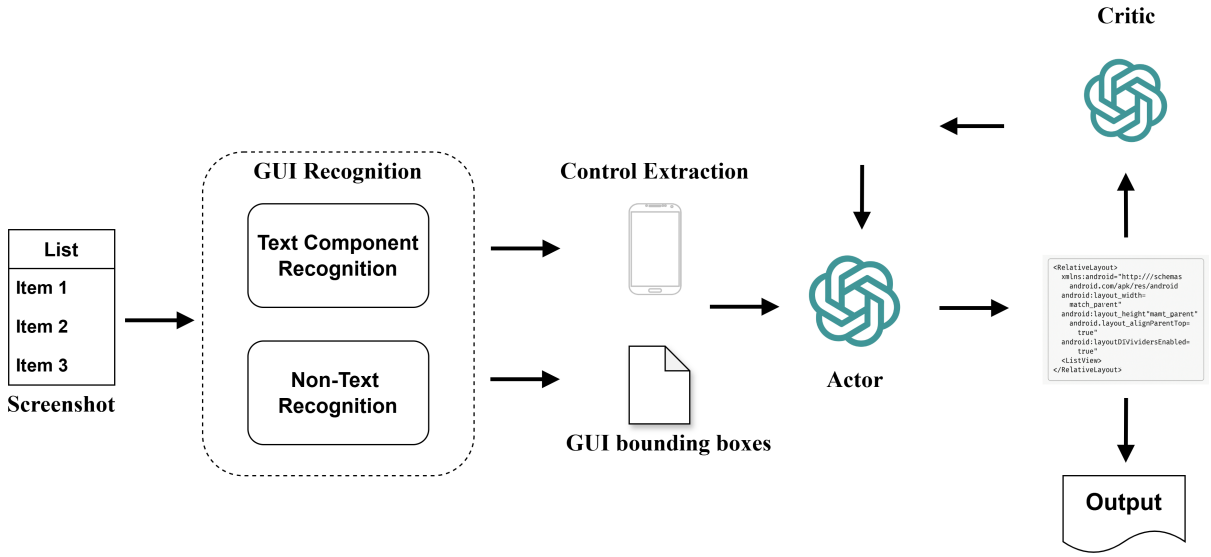

Figure 1: Overall framework of our hierarchical LLM-agent pipeline for Android GUI code generation. The system first performs component-level recognition from GUI screenshots, then delegates structured code generation and refinement to a team of collaborating agents.

toward a final objective. While powerful, these systems also face issues such as error accumulation and objective drift (Gravitas, 2023).

To address quality control, Self-Refine (Madaan et al., 2023) allows the same model to act as both generator and critic, refining its output through iterative self-feedback without additional training. This process improves result quality over one-shot generation across tasks.

These agentic capabilities align closely with the demands of Android GUI code generation. Transforming a screenshot into nested, layout-constrained XML code requires visual interpretation, structural planning, and iterative correction. LLM agents can decompose the task into manageable units (e.g., container-by-container), verify layout consistency, and refine outputs. We build on this paradigm to introduce an LLM-driven approach for Android UI generation that incorporates planning and self-refinement to address the long-horizon nature of GUI layout synthesis.

3 Problem Formulation

We formalize the *UI-to-Code* task for Android GUIs as a conditional generation problem. The **input** is a screenshot image I of an Android application's user interface. The **output** is a structured Android `XML` layout L that accurately reflects the visible UI hierarchy in I. The goal is to generate L', a predicted layout code that approximates the ground-truth L^* corresponding to I. Formally, the task is to learn a function $f : I \to L$ such that $L' = \arg\max_L \mathbb{P}(L \mid I)$.

3.1 Hierarchical Layout Tree

We represent L as a hierarchical layout tree $T = (V, E)$, where each node $v \in V$ denotes a UI component (e.g., `Button`, `TextView`), and each edge $e \in E$ encodes a parent–child relationship. This tree structure naturally maps to XML nesting. For instance, a `LinearLayout` with a `Button` and a `TextView` corresponds to a parent node with two children.

Let $T^* = (V^*, E^*)$ denote the ground-truth layout tree, and $T' = (V', E')$ the tree derived from generated code L'. While each node carries additional attributes (e.g., size, color), we primarily focus on the structural correctness of T' with respect to T^* (Beltramelli, 2017).

3.2 Objective: Structural Similarity

Our objective is to minimize the structural discrepancy between T' and T^*. We use the **tree edit distance** (TED) as the evaluation metric:

$$\text{TED}(T', T^*) = \min \#\text{operations to transform } T' \text{ into } T^*,$$

where operations include node insertion, deletion, and substitution. A lower TED indicates higher structural fidelity.

3.3 Common Error Types

Despite recent progress, generated layouts frequently contain structural errors. We identify three

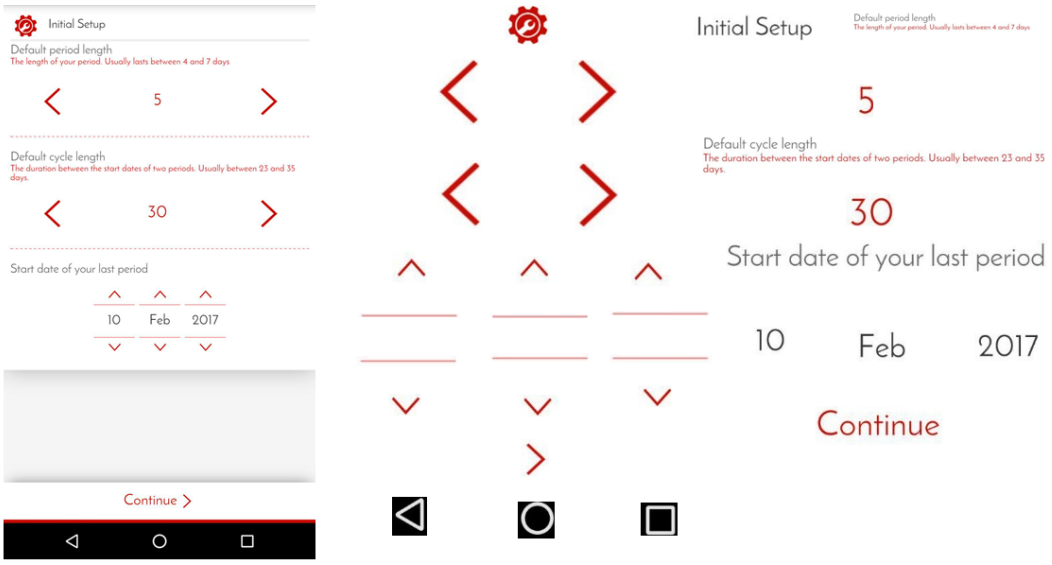

Figure 2: Example of GUI component recognition. The system detects and localizes visual elements—such as buttons, text fields, and icons—from the input screenshot, serving as the foundation for building a hierarchical layout tree.

typical categories:

- **Element Omission:** Missing components that are visible in I, such that $V' \subsetneq V^*$.
- **Element Distortion:** Incorrect component type or attributes (e.g., misclassifying a `TextView` as a `Button`).
- **Incorrect Arrangement:** Misplaced elements due to wrong nesting or ordering, leading to edge errors in E'.

These issues degrade layout quality. Omission leads to missing nodes, distortion introduces semantic errors, and arrangement mistakes break hierarchy. Our method aims to reduce such discrepancies and produce a layout L' where $T' \approx T^*$.

4 Approach

As illustrated in Figure 1, we propose a hierarchical pipeline that generates native Android XML layout code from GUI screenshots via LLM agent collaboration. The framework consists of three main stages: component recognition, agent-based code generation, and reinforcement-based iterative refinement (Madaan et al., 2023). This design enables structured decomposition of the UI-to-code task and progressive improvement of output quality.

4.1 GUI Component Recognition

To extract a structured representation of the interface, we design a dual-branch component recognition network that separately detects textual and non-textual elements in Android GUI screenshots (Deka et al., 2017b). This module provides the layout-aware visual grounding for subsequent code generation, as illustrated in Figure 2.

Text Component Detection. We apply the EAST detector (Zhou et al., 2017), a fast and accurate FCN-based model for oriented scene text detection. EAST directly predicts rotated bounding boxes, making it well-suited for parsing structured textual elements (e.g., labels, prompts) in GUI images. Compared with heavier models such as CRAFT (Baek et al., 2019), EAST offers a good balance of speed and precision for mobile UI domains.

Non-Text Component Detection. We adopt a structure-aware detection strategy inspired by (Chen et al., 2020), focused on layout block extraction and hierarchy reconstruction. The process includes:

1. **Block Segmentation:** Convert input frame to grayscale and apply flood-fill to extract uniform regions R_i.
2. **Edge Map Generation:** Compute structural

gradient magnitude $M_{\text{grad}} = \sqrt{G_x^2 + G_y^2}$ and binarize to obtain component boundaries.

3. **Component Labeling:** Extract connected components and record bounding boxes and center coordinates.

4. **Structure Refinement:** Merge overlapping regions, filter out small components, and build parent-child relationships via spatial containment.

This network operates frame-independently and yields a hierarchical component tree $T = (V, E)$ that reflects both the local geometry and global organization of the UI. It serves as the structural basis for downstream layout code generation.

4.2 LLM Agent Collaboration

To generate structured layout code from the component tree, we design a modular LLM agent framework where multiple specialized agents collaborate to complete the UI-to-code task. Inspired by multi-agent paradigms such as ReAct (Yao et al., 2022) and Toolformer (Schick et al., 2023), we assign distinct roles to agents that operate on subtrees and exchange intermediate outputs via shared memory.

The agent system consists of three key modules:

- **Structure Agent:** Traverses the initial component tree T^{init}, enriching each node with contextual metadata such as container type, orientation, and depth.

- **Local Code Agent:** For each subtree rooted at a UI component, generates XML code fragments based on localized spatial features, inferred widget types, and parent layout hints.

- **Assembly Agent:** Aggregates code snippets from all local agents, merges them into a global layout, and performs consistency checks (e.g., tag closure, nesting order, missing parents) to reconstruct a complete XML layout file.

Each agent operates with partial knowledge of the layout tree but contributes collaboratively to a globally coherent solution. Intermediate messages (e.g., node annotations, XML templates) are passed through a shared memory buffer, enabling coordination without central control. This decomposition reduces the complexity of one-shot generation and improves layout fidelity by embedding structural priors into the generation process.

Algorithm 1: GUI Component Recognition Pipeline

Input: GUI image $I \in \mathbb{R}^{H \times W \times 3}$
Output: Component set $\mathcal{C}$, layout tree $T = (V, E)$

`// Text Component Detection (EAST)`
$\mathcal{C}_{\text{text}} \leftarrow \texttt{EAST}(I)$
`// Non-Text Component Detection (Structure-Aware)`
Convert I to grayscale image G ;
$B \leftarrow \texttt{FloodFill}(G, \epsilon)$; `// Block segmentation`
Compute gradients G_x, G_y and $M_{\text{grad}} = \sqrt{G_x^2 + G_y^2}$;
Apply threshold τ to obtain binary map M_{bin} ;
Extract connected components $\mathcal{C}_{\text{non-text}}$ from M_{bin} ;
Merge overlapping/adjacent regions based on IoU ;
Filter out regions with area $< A_{\text{min}}$;
`// Hierarchical Tree Construction`
$\mathcal{C} \leftarrow \mathcal{C}_{\text{text}} \cup \mathcal{C}_{\text{non-text}}$;
Initialize tree $T = (V, E)$ with virtual root node ;
foreach $c_i \in \mathcal{C}$ **do**
  foreach $c_j \in \mathcal{C}, j \neq i$ **do**
    if $\texttt{Contained}(c_j, c_i)$ ***and*** *no* c_k *satisfies* $c_j \subset c_k \subset c_i$ **then**
      Add edge $(c_i \rightarrow c_j)$ to E
  if c_i *has no parent* **then**
    Add edge $(\text{root} \rightarrow c_i)$ to E
return $\mathcal{C}, T$

4.3 Reinforcement-Based Iterative Optimization

Although the initial layout prediction L' produced by the LLM agent system often captures the overall structure, it may still contain semantic or hierarchical errors—such as missing components, misnesting, or incorrect widget types. To address these issues, we adopt a self-refinement loop that iteratively improves L' through reward-guided revision.

After generating an initial layout L', we evaluate its quality using structural and component-level metrics, including Tree Edit Distance (TED), token-level accuracy, and widget-level precision/recall. We define a reward function $R(L') \in [0, 1]$ to

quantify its similarity to the ground truth layout L^*, combining multiple metrics:

$$R(L') = \lambda_1 \cdot \text{TED}(T', T^*)^{-1} + \lambda_2 \cdot \text{F1}_{\text{widget}}(L', L^*) + \lambda_3 \cdot \text{Acc}_{\text{token}}(L', L^*), \quad (1)$$

where $\lambda_1 + \lambda_2 + \lambda_3 = 1$ are hyperparameters.

The layout is then passed back into the agent loop with a feedback summary, containing structured error descriptions (e.g., "*Button X missing*", "*Incorrect parent for TextView Y*"). Each agent uses this signal to revise its assigned region, producing an updated layout L'_{t+1}. The process repeats for a fixed number of iterations or until convergence:

$$L'_{t+1} = \texttt{Refine}(L'_t, \texttt{Feedback}(L'_t, L^*)).$$

This iterative refinement mimics recent self-feedback frameworks (Madaan et al., 2023), enabling agents to learn from their own errors without additional supervision. In practice, we observe that even two refinement steps significantly boost structural accuracy and component recall.

5 Experiment

We conduct comprehensive experiments to evaluate the effectiveness of our hierarchical LLM-agent framework for Android GUI code generation. This section details the experimental setup—including datasets, evaluation metrics, and baselines—as well as the results and analysis that demonstrate the advantages of our approach.

5.1 Dataset

Unlike HTML, Android XML layouts cannot be rendered or compiled in isolation without integration into an app environment. Consequently, we reformulate the generation task as predicting a hierarchical component tree from GUI screenshots.

Our experiments are conducted on a manually curated subset of the Rico dataset (Deka et al., 2017b), which provides paired screenshots and view hierarchies from real-world Android applications. While Rico is widely used, it contains known annotation errors. We therefore reference two improved derivatives—Fixco and Clay—that offer structural corrections. Additionally, UI2Code contributes supplemental screenshot–layout pairs, and DeclarUI provides unlabeled GUI images. For consistency and annotation quality, our main evaluation is based on a hand-verified Rico subset, with supplementary tests on Clay.

5.2 Evaluation Metrics

Since Android XML layouts cannot be directly rendered or visually compared, we adopt structure-aware automatic metrics to evaluate the generated layout code. Specifically, we report five complementary metrics:

- **BLEU** (Papineni et al., 2002): Measures n-gram overlap between the predicted XML and the ground-truth code.
- **Tree Edit Distance (TED)**: Quantifies the structural similarity between the predicted layout tree and the reference hierarchy. Lower is better.
- **Precision**: The proportion of correctly predicted components among all predicted ones.
- **Recall**: The proportion of correctly predicted components among all ground-truth components.
- **Accuracy**: Token-level correctness of the generated XML layout.

These metrics collectively capture both structural fidelity and semantic correctness of the predicted layouts.

5.3 Baselines

We compare our approach against three representative baselines spanning both traditional and LLM-based paradigms:

- **Pix2Code** (Beltramelli, 2017): A pioneering CNN-RNN model that generates UI code from a single screenshot. It serves as a classical data-driven baseline for UI-to-code generation.
- **Direct Prompting (GPT-4o 1-shot):** The screenshot and task instruction are fed directly into GPT-4o in a single prompt, simulating the upper-bound performance of zero-shot or one-shot generation without decomposition.
- **Chain-of-Thought Prompting:** We prompt GPT-4o using a step-by-step reasoning format to encourage structured layout generation, testing the model's ability to plan before producing code.

All baselines are evaluated using the GPT-4o model (release 2024-11-20) to ensure consistent capacity across different prompting strategies.

Table 1: Main results on BLEU, Tree Edit Distance (TED), and component-level metrics. Higher BLEU, Precision, Recall, and Accuracy are better; lower TED is better.

Method	BLEU ↑	TED ↓	Precision ↑	Recall ↑	Accuracy ↑
Pix2Code	27.4	9.3	0.60	0.50	0.20
GPT-4 (Direct Prompt)	64.0	5.0	0.81	0.75	0.35
GPT-4 (CoT Prompt)	68.2	4.5	0.83	0.80	0.40
Ours (Full)	**79.8**	**3.2**	**0.92**	**0.95**	**0.60**

5.4 Experiment Results and Discussions

Main Results Table 1 presents a comparison between our method and several strong baselines, including traditional UI-to-code models (e.g., Pix2Code) and recent LLM-based prompting strategies (direct and CoT). Across five key metrics—BLEU, Tree Edit Distance (TED), Precision, Recall, and Accuracy—our approach consistently outperforms all competing methods.

Our framework achieves the highest BLEU score (79.8), indicating strong token-level alignment with ground-truth code, and the lowest TED (3.2), reflecting superior structural fidelity. Additionally, it yields the best component-level performance, recovering 95% of UI elements (Recall) while maintaining 92% Precision.

Analysis The performance gains can be attributed to two core design factors: (1) the use of GUI segmentation reduces visual clutter and improves component recognition, and (2) the multi-agent architecture promotes modular, context-aware code generation. Compared to one-shot prompting, our approach better preserves nested layouts and mitigates common errors such as element omission or misplacement. The refinement loop further enhances correctness by enabling iterative self-improvement based on structured feedback.

Summary These results demonstrate the effectiveness of hierarchical decomposition and agent-based cooperation in improving layout fidelity and component coverage. Our method achieves substantial gains over both traditional and prompting-based baselines, establishing a new performance benchmark for Android GUI code generation.

6 Conclusion

We propose a hierarchical framework for Android GUI code generation from screenshots, powered by a team of collaborating LLM agents. Unlike prior approaches that rely on end-to-end models or single-shot prompting, our method decomposes the task into structured stages—component recognition, agent-based layout generation, and iterative refinement—enabling more accurate recovery of UI hierarchies and better alignment with visual input. Extensive experiments on Android GUI benchmarks demonstrate that our approach significantly outperforms classical models and prompting-based LLM baselines across lexical, structural, and component-level metrics. These results highlight the effectiveness of structured decomposition and multi-agent planning in bridging visual understanding and code generation for mobile interfaces.

Limitations

Dependence on Proprietary LLM APIs. Our implementation relies on commercial large language model APIs (e.g., GPT-4) for code generation. This dependence may limit deployment in offline or low-resource settings, as access to internet-connected and paid services is required. It also raises concerns about reproducibility and long-term accessibility in non-commercial or constrained environments. **Single-Screen Focus.** The current system only supports static, single-screen GUI screenshots. It does not account for multi-screen navigation flows or UI state transitions. As such, it is limited in modeling dynamic behaviors commonly seen in real-world applications, where UI logic spans across multiple pages or components. **Requirement for Ground-Truth Supervision.** The reinforcement-based refinement step assumes access to ground-truth labels, such as reference XML layouts. While effective in controlled settings, this assumption does not hold in real-world deployments where annotated layouts are often unavailable, making the refinement process less applicable in zero-shot or unlabeled scenarios.

References

Youngmin Baek, Bado Lee, Dongyoon Han, Sangdoo Yun, and Hwalsuk Lee. 2019. Character region awareness for text detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Tony Beltramelli. 2017. pix2code: Generating code from a graphical user interface screenshot. In *arXiv preprint arXiv:1705.07962*.

Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2018. Object detection for graphical user interface: Old fashioned or deep learning or a combination? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object detection for graphical user interface: Old fashioned or deep learning or a combination? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Biplab Deka, Zifeng Huang, Kevin Franzen, John Hibschman, Michael Afergan, Yang Li, and Ranjitha Kumar. 2017a. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST)*.

Biplab Deka, Zifeng Huang, Kevin Franzen, John Hibschman, Michael Afergan, Yang Li, and Ranjitha Kumar. 2017b. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST)*.

Significant Gravitas. 2023. Autogpt: An experimental open-source attempt to make gpt-4 fully autonomous. https://github.com/Torantulino/Auto-GPT.

Yujian Liu, Yuxian Zhou, Xiangning Lin, Mu Li, Xu Sun, and Graham Neubig. 2021. Pico: Parameter-efficient image-to-code generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Aman Madaan, Xiang Lin, Satyapriya Singh, Xisen Liu, Huan Yu, Graham Neubig, Shinn Yao, and Pengfei Liu. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

Kevin Moran, Zhilei Lin, Carlos Bernal-Cardenas, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 540–550.

Anh Tuan Nguyen, Chris Piech, and Tien N. Nguyen. 2015. Remaui: Automatically extracting ui code from mobile app screenshots. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 72–83.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL)*, pages 311–318. ACL.

Timo Schick, Arun Dwivedi-Yu, Saba Hosseini, Frederik Raue, Rahul Singh, Antoine Bosselut, Marc'Aurelio Ranzato, Tal Linzen, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.

Bowen Si, Haoyu Zhang, Yutian Li, Chenguang Zhu, Mu Li, and Wei Sun. 2024. Depth matters: Scaling ui-to-code models to complex layouts. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Xin Wan, Yufei Zhang, Jiahao Chen, and Zhiyuan Liu. 2024. Segment-then-prompt: Accurate web ui code generation with layout-aware prompting. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Shinn Yao, Jinyi Zhao, Dian Yu, Izhang Zhao, Kunwoo Park, Esin Durmus, and Yejin Choi. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. 2017. East: An efficient and accurate scene text detector. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5551–5560.