Minuku: Detecting Diverse Display Issues in Mobile Apps with Small-scale Dataset

1st Yongxiang Hu
College of Computer Science and Artificial Intelligence
Fudan University
Shanghai, China
yongxianghu23@m.fudan.edu.cn

2nd Ke Liu
College of Computer Science and Artificial Intelligence
Fudan University
Shanghai, China
keliu24@m.fudan.edu.cn

3rd Hailiang Jin
Meituan
Shanghai, China
jinhailiang@meituan.com

4th Shiyu Guo
Meituan
Shanghai, China
guoshiyu03@meituan.com

5th Juxing Yuan
Meituan
Beijing, China
yuanjuxing@meituan.com

6th Xin Wang
College of Computer Science and Artificial Intelligence
Fudan University
Shanghai Key Laboratory of Intelligent Information Processing
Shanghai, China
xinw@fudan.edu.cn

7th Yangfan Zhou
College of Computer Science and Artificial Intelligence
Fudan University
Shanghai Key Laboratory of Intelligent Information Processing
Shanghai, China
zyf@fudan.edu.cn

***Abstract*—User interface (UI) display issues, such as widgets occlusion, missing elements, and screen overflow, are emerging as a non-negligible source of user complaints in commercial mobile apps. However, existing automated testing tools typically rely on a vast amount of high-quality training data, making them cost-ineffective for industrial practice. Given that display issues are intuitively recognizable by humans, their diverse appearances can be abstracted by the violation of human commonsense expectations of UI appearance. Therefore, this paper proposes to reduce data requirements in display issue detection through commonsense simulation. Although leveraging large vision-language models (VLMs) to replicate human visual ability looks straightforward, off-the-shelf VLMs lack task-specific knowledge of UI designs and display correctness. To address this, we fine-tune a VLM to learn what constitutes an expected display and to reason potential display issues. This approach is termed as `Minuku`, an industrial data-efficient UI display issue detector. We evaluate the design effectiveness of `Minuku` via a set of ablation experiments. Moreover, real-world deployments in one of the largest E-commerce app providers further demonstrate that `Minuku` can effectively detect 40 previously unknown UI display issues and significantly reduce manual effort in industrial settings.**

***Index Terms*—Automatic UI Testing, Display Issue, Mobile Apps**

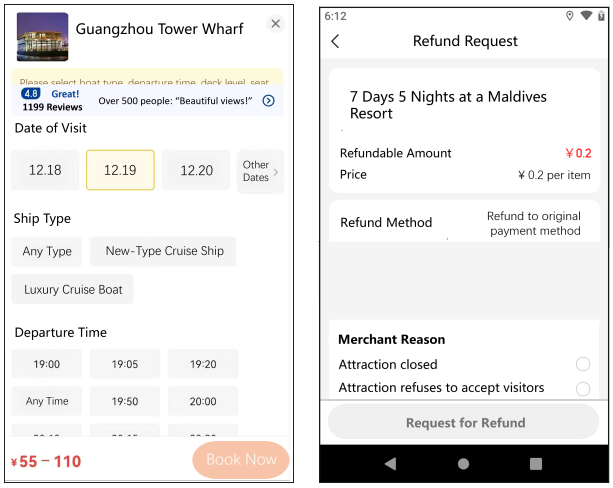
I. Introduction

Driven by increasingly intense competition between commercial mobile apps, anomalies of user interface (UI) that disrupt user experience are receiving growing attention from industry practitioners [1], [2]. While extensive research has focused on detecting app crashes [3]–[9] and incorrect responses [10]–[15], UI display issues, *e.g.*, button misalignment, missing images, and title truncation, were previously underexplored [16], [17]. As critical defects such as app crashes have declined in mobile apps, display issues now account for an increasing proportion of user complaints [18], suggesting the inadequacy of existing testing practices.

Existing tools for display issues are typically based on neural network (NN) detectors [17], [19] or heuristic rules [20], [21]. However, their effectiveness in real-world commercial apps remains limited [16]. Specifically, the visual diversity across display issues (as shown in Figure 1) often leads to considerable missed detections when using fixed heuristic rules [22]. Although NN-based detectors can handle such diversity, they rely on a vast amount of high-quality training data. However, such data inevitably requires substantial manual efforts, making these methods long regarded as cost-ineffective in industrial practice. How to design NN-based detectors to handle such visual diversity given only a small amount of training data is a critical task in industry practice.

While existing tools struggle with the diversity of display issues, these issues typically manifest as violation of human commonsense expectations of UI appearance. As a result, in practice, human users in contrast can easily identify these issues even for previously unseen UI. Therefore, given the increasing use of large vision-language models (VLMs) to replicate human capabilities [23], [24], leveraging these models to acquire human-like commonsense and automated detection of display issues presents a natural solution. However, due to

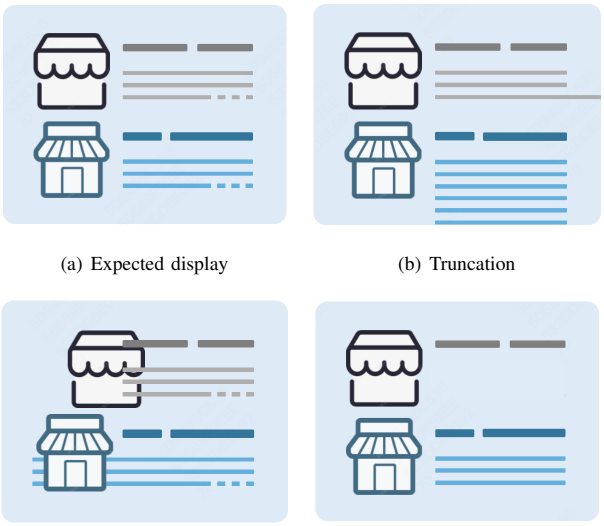
2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00270



(a) *Occlusion* where rating blocks instructions.

(b) *Blank* from missing elements.

Fig. 1: Example of UI display issue.



(a) Expected display

(b) Truncation

(c) Occlusion

(d) Blank

Fig. 2: Expected display and categorization of display issues.

their lack of task-specific knowledge of UI designs and display correctness, they are not inherently capable of detecting display issues. To address this, we fine-tune a general-purpose open-source VLM using semantically annotated screenshots of both correct and anomalous UI displays. This enables the VLM to understand what constitutes an expected display and to reason potential display issues. To the best of our knowledge, this is the first work to detect UI display issues by simulating human commonsense. We term this approach as `Minuku`.

We specifically solve two challenges in applying VLMs to UI display issue detection. The first is that the small-scale training data is prone to leading models to memorize specific screenshots instead of learning the underlying commonsense. To mitigate this, inspired by DeepSeekMath [25], we incorporate an additional reinforcement training phase using group relative policy optimization (GRPO) after standard supervised fine-tuning (SFT), to help the model better capture the underlying logic. The second challenge is that semantic annotation (*i.e.*, reasoning the commonsense violation or alignments in each UI screenshot), even for small datasets, remains particularly labor-intensive. To address this, we use synthesized anomaly screenshots to automatically provide guidance for unsupervised VLMs to generate initial drafts of UI display analysis. This approach substantially minimizes manual effort by changing data annotation to verification.

Comprehensive experiments are conducted to evaluate the design effectiveness of `Minuku`. Specifically, an ablation study is performed to evaluate the contributions of each component in `Minuku`. The result shows `Minuku` significantly outperforms all its variants, demonstrating the effectiveness of its fine-tuning procedure. Moreover, constructed on an open-source VLM with 7 billion parameters, `Minuku` outperforms existing NN-based detectors and state-of-the-art, un-finetuned closed-source VLMs. It achieves the highest F_1 of 80% while preserving a low false positive rate of 10%. We further demonstrate our field deployment experience of `Minuku` to test real-life shopping apps from Meituan, one of the largest E-commerce app providers serving over 600 million users. Our practice shows that it has successfully detected 40 previously unknown display issues from more than 4,500 screenshots and significantly reduced manual effort in UI display testing.

We summarize the contributions of this paper as follows.

- We share our field observations that UI display issue testing has long been cost-ineffective in industrial practice. We propose that the key obstacle is the high reliance of existing methods on large volumes of high-quality training data.
- By analyzing how humans identify display issues, we propose that the large-scale data can be replaced by the simulation of human commonsense expectations of UI appearance. We make this idea feasible by fine-tuning a general-purpose open-source VLM using semantically annotated screenshots.
- Extensive experiments show that our method outperforms existing approaches under small-scale data settings. Moreover, real-world deployment on 4,777 screenshots from commercial mobile apps further demonstrates its industrial feasibility.

The rest of the paper is organized as follows. Section II introduces categorization of display issues and highlights the lack of sufficient testing in industrial practice. In Section III, we demonstrate the design of `Minuku` by proposing each of its three modules. Then, three questions are answered in Section IV to evaluate `Minuku`'s performance and usability. Section V elaborates our practical cases for the deployment of `Minuku`. Threats to validity and future work are discussed in Section VI. Finally, the paper is concluded in Section VIII.

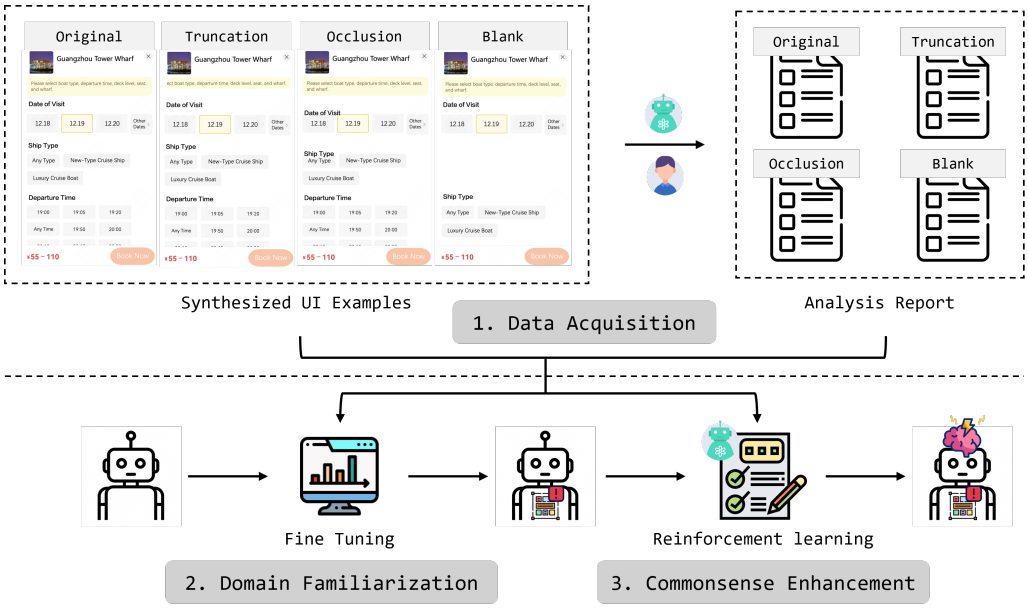


Fig. 3: Overview of Minuku.

II. Preliminary Study

In this section, we demonstrate the background and motivation of Minuku by presenting our practical experience in Meituan. Specifically, three months of historical UI display issues from Meituan have been gathered to study their manifestations. Moreover, testing practice is investigated to show the shortcomings of existing methods.

A. Characterization of Display Issues

To align with practical needs, we collected three months of reported display issues from Meituan. The 25 reported display issues are analyzed and classified into four categories. As shown in Figure 2, this paper focuses on three types of inter-element display issues: *Truncation* (elements exceeding parents' bounds), *Occlusion* (elements blocking one another), and *Blank regions* (unintended empty areas).

Text-related anomalies, such as truncation or overlength, are not the focus of this study, as they are typically addressed by more specialized testing techniques [1], [20]. Moreover, we acknowledge that the types of display issues that receive attention may vary across different contexts or time periods. In Section VI, we will further discuss the applicability of Minuku to other types of display anomalies beyond the three categories specified in this study.

B. Practical Detection Practice

UI display issue testing in Meituan is typically performed either manually or using object detection models. For instance, a YOLO-based [26] detection model is trained on a large volume of annotated screenshots. However, test engineers soon observed that the iteration of UI designs led to a significant drop in the model's output accuracy. Unfortunately, the typical frequent UI iteration creates a recurring loop of data collection, annotation, and model retraining, resulting in an unsustainable long-term maintenance burden. Consequently, model updates are frequently delayed, leading to outdated models with reduced accuracy and further manual verification.

Therefore, in current practice, existing cost-ineffective detection methods are not widely adopted, and as a result, a large portion of UI display issues in Meituan are still manually discovered. Testers rely on their experience and visual intuition to spot problems that deviate from the expected UI appearance. While this approach is flexible and adaptable to diverse UI designs, it is inherently unscalable and inefficient. In practice, it is common for human testers to manually cover a rather limited subset of the screenshots, especially when under time pressure. These omissions have led to a growing number of user complaints about display issues, undermining the overall user experience.

To this end, with the aim of reducing manual effort in detecting display issues, this paper aims to enable NN-based detectors to reason about UI display correctness in a human-like way. From the end-user's perspective, display issues always stem from violations of common-sense visual expectations, such as when a component is unexpectedly missing or misplaced. By learning the commonsense of human UI comprehension, a detection tool can be applied to a wide range of display issues without requiring exhaustive data collection, annotation, or frequent maintenance.

III. Method

A. Overview

Figure 3 shows the overview of Minuku, the UI display issues detection tool we designed for mobile apps. Given

You will receive a screenshot of an App.
Based on the known result, generate the reasoning process.
===Known Result===
There are UI display issues within the marked area. There are no problems elsewhere on the screenshot.
UI displays correctly.
===Analysis Criteria===
UI Display Issue: Any situation that clearly affects user experience and usage is considered a UI display issue.
Problem Identification: You need to find all UI display issues within the marked area.
Red Box Explanation: The box mark is just for helping locate the issue. Do not mention the red box or use it to locate issues in your answer.
Notice: For areas that are prone to misjudgment, please provide necessary analysis and explanation.
Description Requirements: Use adjacent elements in the screenshot to help describe the location of the UI issue.
Analysis Requirements: Pretend you do not know the known result. Your answer must not mention the known result. Simulate your reasoning process to arrive at the known result.
Answer Content: Output your reasoning process within the <think> tag, and output your final answer within the <answer> tag. Do not provide suggestions for improvement. You answer should in <think></think> <answer></answer> format.

Fig. 4: Prompt for UI comprehension-based annotation.

that display issues detection is an unfamiliar task for general-purpose VLMs, `Minuku` enhances their capability through three steps: data acquisition, domain familiarization, and commonsense enhancement.

Specifically, data acquisition is responsible for providing analysis for screenshots with and without display issues. Subsequently, in the domain familiarization step, a general-purpose VLM is prompted with a screenshot to detect display issues, completing the template: <answer> <reason>. During this process, `Minuku` gains a basic UI understanding through Supervised Fine-Tuning (SFT), enabling it to gain an initial grasp of UI design and display issues. Finally, to further enhance the human-like commonsense of display issues, we employ reinforcement learning with General Reinforcement Preference Optimization (GRPO) [25], which refines the model's detection accuracy.

B. Data Acquisition

Training data is acquired following a three-step process. First, as discussed in Section II-A, we define recognition targets of display issues as *Truncation*, *Occlusion*, and *Blank Regions*. Then, we construct synthetic UI examples that exhibit these issues. Finally, we augment the data with semantic annotations in the form of commonsense-based explanations, initially generated with a VLM and manually verified.

Since the amount of historical data is limited and may deviate from current UI design trends, we manually collected and synthesized training screenshots based on the latest version of apps in Meituan. For *Truncation*, we simulate partial clipping

Given an App screenshot, analyze whether there are any UI display issues.
===Please follow the instructions below===
1. Task Definition: Display issues refer to obvious display problems that affect user experience, such as overlapping components, occlusions, text overflowing boundaries, or abnormal blank modules. The types of display issues are not limited to these examples.
2. Tips: Use adjacent and clearly visible elements to help describe the location of the problem.
3. Answer format: Fill your thought process in the <think></think> tag, and put the final answer in the <answer></answer> tag.

Fig. 5: Task description for fine-tuning.

by randomly shifting a UI component and covering part of it with a background-colored block. As for *Occlusion*, we simulate overlapping by randomly shifting a UI component to overlap with an adjacent one. Finally, for *Blank Regions*, a background-colored block is created to block an entire UI component to create unintended empty spaces.

While recognizing display issues is important, it is equally essential for the model to know what a correct UI display looks like. To this end, we adopt a dual-sample generation strategy: for each synthetic abnormal example, we retain its original screenshot (*i.e.*, the normal version) as the training data. This design encourages the model to understand the distinction between visually correct and anomalous UI screenshots.

To generate training annotations for these screenshots, we designed a VLM-augmented annotation pipeline to generate semantic explanations. Specifically, as shown in Figure 4, we formulated two task-specific prompts depending on whether the input UI screenshot contains a display issue. For positive cases (*i.e.*, screenshots with display issues), the issue area is indicated by a red box, and the model is asked to explain its visual abnormality based on its visual comprehension and logical reasoning. Since the content within the issue area may be unclear or visually distorted, the model is instructed to refer to adjacent, correctly rendered components (*e.g.*, "the area to the left of the confirm button") to describe the issue location. For negative cases, we prompt the model to reason about why the given screenshot should be considered correct while explicitly addressing UI elements that might otherwise be misinterpreted as display issues.

The annotation format is borrowed from DeepSeekMath [25] with <think> and <answer> sections. The <think> section contains the UI analysis and comprehensive steps, and the <answer> section provides the final judgment on whether a display issue exists. This structure helps the model distinguish between normal and abnormal UI displays and makes the reasoning process explicit.

In addition to the VLM-based pre-annotation, all annotated samples were manually reviewed by three authors of this paper, each with over six months of experience in UI testing for Meituan. Specifically, each reviewer independently verified

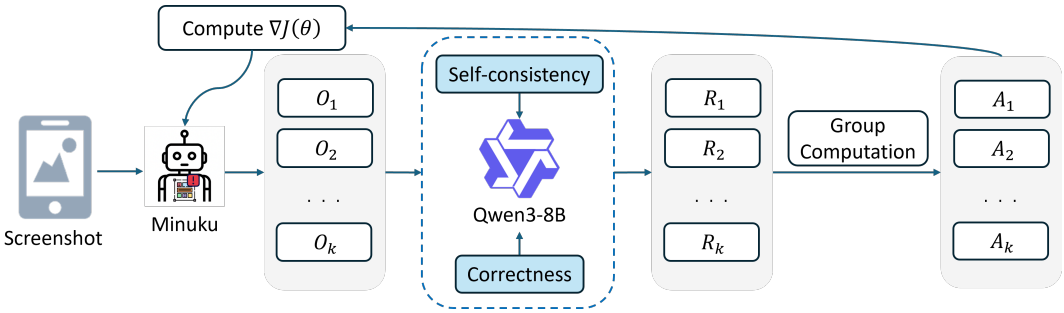


Fig. 6: Workflow of commonsense enhancement procedure in Minuku.

two-thirds of the samples, evaluating both the reasoning (in the <think> block) and the final judgment (in the <answer> block) for factual correctness and logical consistency. In cases of disagreement, the other author was consulted for discussion and final arbitration. In practice, about 8% of the VLM's pre-annotations, especially in the <think> block, required manual correction.

C. Domain Familiarization

Comparing the visual understanding capability of different open-source VLMs [27], [28], Qwen2.5VL-7B-Instruct [29] is selected as Minuku's base model due to its small scale and outstanding performance. However, as a general-purpose model, it lacks domain-specific knowledge in UI display issues. Therefore, we employ a display issues identification finetuning task to enable the VLM to quickly acquire relevant domain knowledge. Specifically, given a UI screenshot, the model is instructed to analyze whether it contains display issues, and, if present, describe the problem and its location using natural language. To this end, since our constructed training data provides the reasoning procedure for the final answer, the model can acquire UI knowledge while also learning the underlying explanation.

We acknowledge that UI design and display issues are continuously evolving, and that past bugs cannot fully represent future cases. Therefore, rather than conveniently formulating the task as a classification problem, we design the instruction prompt to be open-ended: we do not constrain the VLM to detect predefined three types of display issues, nor do we require it to provide categorical labels. Instead, as shown in Figure 5, the model is encouraged to identify and describe any visual anomalies that may violate user experience. Then, the difference between the model's output and the annotated data is measured using cross-entropy [30] loss, enabling the parameters optimization.

To improve the model's UI understanding, we designed a two-stage LoRA-based [31] fine-tuning strategy inspired by the training pipeline of Qwen2.5-VL [32], progressively tuning all three components of the model. In the first stage, we freeze the *Large Language Model* and optimize the *Vision Encoder* and *Vision-Language Merger*, aiming to enhance the model's perception of UI screenshots. As for the second stage, we further apply LoRA to all three components of Qwen2.5-VL, thereby improving its vision-language alignment to perceive the human-like commonsense behind display issues.

This process gradually enhances the model's ability from specific visual perception to generalizable semantic comprehension. Furthermore, the advantages of Minuku's two-stage SFT process are further validated by the experimental results in Section IV-B.

D. Commonsense Enhancement

After the SFT phase, the VLM model gains domain-specific knowledge and is able to generate outputs in the required format with a basic understanding of UI design. However, its correctness and logical consistency still lag far behind that of end users, leading to misjudgment and mismatches between the answer and the reasoning. This leads to incorrect explanations for abnormal UIs and false positives on normal ones. Therefore, Minuku further optimises the model's commonsense and reasoning capability to improve its practical usability.

Inspired by recent success in applying GRPO to mathematical reasoning tasks [25], as shown in Figure 6, we explore its effectiveness in improving display issue detection for Minuku. Unlike the token-level optimization used in SFT, reinforcement learning methods such as GRPO make it possible to leverage semantic-oriented reward signals to guide the commonsense and reasoning consistency improvement. Specifically, the model is asked to analyze each screenshot k times, and the semantic difference between outputs and ground truth is measured by another LLM. Specifically, given that both the model's output (denoted as O) and the annotated ground-truth (denoted as G) contain two components: <reason> and <answer>, they can be formalized as follows:

$$O = O^{rea} + O^{ans}$$
$$G = G^{rea} + G^{ans}$$

Then, the reward score for O_i (represented as R_i) is calculated based on the output's consistency and semantic similarity comparing to annotated ground-truth answer:

$$R_i = Cons(O_i^{rea}, O_i^{ans}) + Simi(O_i^{ans}, G_i^{ans})$$

This reward function enables the model to learn the underlying commonsense of UI design rather than linguistic patterns of

There are two descriptions regarding App UI display issues.
Calculate the semantic similarity between the two descriptions, with a score range of 0.0-1.0.
Only output the score.
=== Scoring Criteria ===
0.9-1.0 (Completely Consistent)
The issue type, location, impact, and details are all the same, with only different expressions.
0.7-0.8 (Highly Similar)
The issue location, type, and impact descriptions are consistent, but there are minor wording differences or missing details.
0.5-0.6 (Moderate Similarity)
The issue location is similar or the type is the same but the impact is different, or there is a large difference in the number of issues.
0.3-0.4 (Low Similarity)
Only partially related issues, or the issue types are similar but the locations are unrelated.
0.1-0.2 (Negligible Similarity)
The types are the same but the locations are unrelated, or there is only a single common point.
0.0 (Completely Unmatched)
Contradictory descriptions, or the issue type, location, and impact are completely unrelated.
=== Actual Description ===
{text1}
=== Predicted Description ===
{text2}

Fig. 7: Prompt for reward model.

the annotation, and encourages the improvement of detection accuracy.

Here, `Minuku` leverages the Qwen3-8B [33] to handle the complex tasks of both semantic consistency (function $Cons$) and similarity (function $Simi$) calculation. Specifically, as shown in Figure 7, we provide a prompt outlining the scoring criteria to ensure objectivity and reduce the randomness of the LLM, enabling it to assign rewards according to our predefined rules.

In terms of training configuration, we incorporate several key optimizations from the Decoupled clip and Dynamic sAmpling Policy Optimization DAPO [34]. Specifically, we utilize *Dynamic Sampling* to efficiently manage the training data stream, remove the KL divergence term from the objective function, and apply both *Clip-Higher* and *Token-Level Policy Gradient Loss* in the standard computation of the objective improvement (*i.e.*, $\Delta J(\theta)$) in GRPO. These optimizations collectively enable the model to explore a broader range of solutions and mitigate the disproportionate influence of output length on learning.

IV. Evaluation

In this section, `Minuku`'s effectiveness is evaluated by answering the following questions.

- **RQ1:** What are the contributions of different components in the design of `Minuku`?
- **RQ2:** How effective is `Minuku` in detecting display issues compared to existing methods?
- **RQ3:** To what extent can `Minuku` detect display issues in real-world, practical commercial mobile apps?

RQ1 evaluates the contributions of each component in the design of `Minuku` through an ablation study, examining whether these parts can enhance the model's ability to identify display issues and to what extent they contribute to its overall performance. RQ2 compares the effectiveness of `Minuku` with existing display issue detection methods, assessing its accuracy and ability to generalize across different UI designs. Finally, the practical usability of `Minuku` is validated in RQ3 by recognizing real-life historical UI display issues manually collected from Meituan. Moreover, in Section V, we will present the practical effectiveness of `Minuku` by deploying it to real testing circumstances involving thousands of UI pages in Meituan.

A. Evaluation Setup

1) Benchmark Construction: Although we are not the first to investigate the detection of UI display issues for mobile apps, this area has not been an active research topic in recent years. Given the rapid evolution of UI design for mobile applications, the benchmarks of existing tools have become largely outdated. Therefore, to the best of our knowledge, there is currently no publicly available or suitable benchmark specifically for display issues testing. Consequently, we construct two benchmarks ourselves using screenshots from Meituan 's mobile apps.

Specifically, we selected three main business lines from Meituan: *Dining Voucher*, *Grocery Retail*, and *Admission Ticket*. Each of these business lines has over one million daily active users, features intricate UI designs and diverse functionalities, making them ideal for evaluating the real-world effectiveness of display issue detection. Given that popular commercial apps typically deploy across multiple platforms (*e.g.*, Android, iPhone, mini-programs), variations in UI element display and styles are prevalent across different platforms even for identical pages. Thus, to ensure the objectivity of our test data, the UI screenshot of our benchmark preserved these three sources.

In **RQ1** and **RQ2**, 300 different screenshots are collected from these three different business lines to evaluate the contribution of `Minuku`'s various components and provide a modernized test suite for comparing the accuracy of different display issue detection tools. For each business line, we randomly selected 100 visually correct UI screenshots from the past three months of UI testing logs in Meituan, and manually injected display anomalies into 60 of them. Specifically, we constructed three distinct types of display issues (20 instances for each) by relocating UI elements or adding blank blocks with background colors. The annotation process for these anomaly screenshots follows the same procedure as described in Section III-B. Consequently, the benchmark for RQ1 and RQ2 comprises 300 screenshots, 180 of which contain display issues, including 60 instances for each of the three issue types.

To thoroughly test the real-life feasibility of Minuku, we construct another benchmark for **RQ3** from real historical display issues in Meituan. Specifically, this benchmark consists of 25 real-world display issue screenshots collected from the past three months, along with 50 normal screenshots manually gathered from the latest version of Meituan's apps. In addition to accurately distinguishing between normal and anomalous UI screenshots, the ability to provide explanations for detected issues is also crucial for assessing real-world feasibility. To support this, two of the authors manually annotated each anomalous screenshot with its display issue type and root cause as ground truth. Ultimately, the benchmark for RQ3 consists of 25 real-world display issue screenshots annotated with explanations, accompanied by 50 normal screenshots.

2) Baselines: Since **RQ1** evaluates the effectiveness of Minuku's design, we perform that by comparing it with three other variants. Specifically, to assess whether our fine-tuning procedures involved in Minuku have a positive effect and to what extent they are effective, variants are constructed by progressively removing components from Minuku's architecture. These variants serve as baselines for our ablation study.

First, we remove the final GRPO procedure to obtain the first variant, *VLM-Vis*. This variant includes Minuku's two-stage SFT process tailored for the visual comprehension task. Subsequently, we remove the first stage of the two-stage SFT process (*i.e.*, the Vision Encoder SFT) to obtain the *VLM-SFT* variant, which allows us to evaluate the contribution of Minuku's specific SFT design. Finally, we remove the SFT process entirely to obtain the *VLM* variant, which allows the evaluation of the original model.

In **RQ2**, we evaluate the ability of display issue detection of Minuku by comparing it with existing detection tools. Specifically, due to the limited number of publicly available methods, we selected the classic *OwlEyes* [19] as a representative of CNN-CAM-based detection approaches. As its original training data is currently unavailable, we train it using the same dataset as Minuku to ensure a fair comparison. Besides, given the strong alignment between object detection and display issue detection, we further include *M-YOLO*, the real-life deployed YOLOv8s-based model mentioned in Section II-B, as another representative baseline of NN-based detectors. Additionally, we included advanced, un-finetuned closed-source VLMs, GPT-4.1 [35], Claude-3.7-Sonnet [36], and Gemini-2.5-Pro [37], as further baselines. They will directly answer questions elaborated in Figure 5, to evaluate the effectiveness of Minuku's fine-tuning.

In **RQ3**, we assess Minuku's ability to identify display issues in real-life scenarios by demonstrating its performance on historical issues collected from Meituan. To thoroughly evaluate Minuku's real-life capabilities, we keep the same baselines as in RQ2.

3) Metrics: As for **RQ1** and **RQ2**, due to the large volume of test screenshots, we simplified the data analysis process by directly formulating it as a binary classification problem (*i.e.*, presence or absence of display issues). Given that generative models (*i.e.*,Minuku and its variants) output in natural language, we leverage GPT-4o [38] to assess their semantic correctness. This semantic judgment is based solely on the content of the <answer>. In particular, if the answer contains any ambiguity, it is deemed incorrect and subsequently categorized as either a false positive or a false negative depending on the ground truth.

As for the statistical metrics, we choose the widely used metrics of *Precision*, *Recall*, and F_1. Furthermore, considering the low proportion of display issues in real-world scenarios, we incorporated *FPR* to quantify the manual analysis effort incurred by false positives.

For **RQ3**, despite sharing the same detection task, we introduced one additional statistical metric to reflect real-world usability. One metric assesses the accuracy of baseline explanations (for VLM-based methods) or coordinates (for DNN-based methods) of display issues based on successful recalls (*i.e.*, true positives). The authors manually performed this verification, and the proportion of Minuku's correct explanations was recorded as Acc_{exp}. Furthermore, since RQ3 involves screenshots from the latest app version, we retained *FPR* as an indicator to quantify the practical performance of Minuku in real-world deployment scenarios.

4) Implementation and Configuration: We constructed the training set for Minuku based on 126 normal screenshots collected from the latest version of the Meituan app, following the data synthesis process described in Section III-B. After filtering out unreasonable or inconspicuous anomalies, we retained 296 training screenshots for both fine-tuning and reinforcement learning steps. Although both were manually collected, we ensured that none of the training samples overlapped with the test set used in RQ3.

As for the baselines construction, all variants used in RQ1 share the same training dataset as Minuku for a fair comparison. For RQ2 and RQ3, since the original training data of *OwlEyes* is currently unavailable and unrelated to Meituan 's business, we retrained it on the same dataset as Minuku to ensure a fair comparison. As for *M-YOLO*, considering that YOLO-based approaches typically require a large amount of training data, we directly used the internal Meituan model trained on 1,018 annotated screenshots.

B. RQ1: Contribution of Each Component in Minuku 's Architecture

As shown in Table I, the original VLM model achieves a significant performance improvement over the base model (*i.e.*, *VLM*) after the SFT stage. Notably, the two-stage SFT approach, as applied in *VLM-Vis*, results in a substantial increase in recall compared to *VLM-SFT*, which demonstrates the benefits of separately visual-encoder fine-tuning for screenshot comprehension. However, this improvement in recall is accompanied by a marked rise in the false positive rate. In practical scenarios, where the number of normal UI images far exceeds that of anomalous ones, a false positive rate as high as 0.692 would substantially increase the cost of manual review.

The comparison between Minuku and *VLM-Vis* further demonstrates the advanced architecture of Minuku. Minuku

TABLE I: Performance comparison of Minuku and variants

Baselines	Dining Voucher				Grocery Retail				Admission Ticket				Avg.			
	Pre.	Rec.	F_1	FPR	Pre.	Rec.	F_1	FPR	Pre.	Rec.	F_1	FPR	Pre.	Rec.	F_1	FPR
VLM	0.727	0.267	0.390	0.150	0.556	0.250	0.345	0.300	0.472	0.283	0.354	0.475	0.585	0.267	0.367	0.308
VLM-SFT	0.837	0.683	0.752	0.200	0.759	0.733	0.746	0.350	0.767	0.550	0.641	0.250	0.788	0.656	0.716	0.267
VLM-Vis	0.610	**0.833**	0.704	0.800	0.667	**0.900**	0.766	0.675	0.676	**0.833**	0.746	0.600	0.651	**0.856**	0.740	0.692
Minuku	**0.938**	0.750	**0.833**	**0.075**	**0.898**	0.733	**0.807**	**0.125**	**0.907**	0.650	**0.757**	**0.100**	**0.914**	0.711	**0.800**	**0.100**

TABLE II: Performance comparison of Minuku and baselines

Baselines	Dining Voucher				Grocery Retail				Admission Ticket				Avg.			
	Pre.	Rec.	F_1	FPR	Pre.	Rec.	F_1	FPR	Pre.	Rec.	F_1	FPR	Pre.	Rec.	F_1	FPR
OwlEyes	0.517	0.500	0.508	0.700	0.527	0.483	0.504	0.650	0.491	0.450	0.470	0.700	0.478	0.512	0.494	0.683
M-YOLO	**0.950**	0.633	0.760	**0.050**	**0.923**	0.600	0.727	**0.075**	**1.000**	0.500	0.667	**0.000**	**0.958**	0.578	0.721	**0.042**
GPT-4.1	0.813	0.433	0.565	0.150	0.839	0.433	0.571	0.125	0.769	0.333	0.465	0.150	0.807	0.400	0.535	0.142
Claude-3.7	0.897	0.433	0.584	0.075	0.824	0.467	0.596	0.150	0.800	0.267	0.400	0.100	0.840	0.389	0.532	0.108
Gemini-2.5	0.633	**0.950**	0.760	0.825	0.624	**0.967**	0.758	0.875	0.648	**0.950**	**0.770**	0.775	0.635	**0.956**	0.763	0.825
Minuku	0.938	0.750	**0.833**	0.075	0.898	0.733	**0.807**	0.125	0.907	0.650	0.757	0.100	0.914	0.711	**0.800**	0.100

TABLE III: Historical issues detection performance comparison

Baseline	#Rec.	#Rec_{dis}	Recall	Acc_{dis}	Pre.	F_1	FPR
OwlEyes	18	4	0.720	0.222	0.375	0.493	0.600
M-YOLO	13	9	0.520	0.692	0.684	0.591	0.120
Gemini-2.5	**25**	**18**	**1.000**	**0.720**	0.403	0.575	0.740
Minuku	20	14	0.800	0.700	**0.952**	**0.870**	**0.020**

[1] "#Rec." denotes the number of recognized issues.
[2] "#Rec_{dis}" indicates the number of display issues correctly explained or accurately located, and Acc_{dis} refers to its proportion relative to "#Rec.".

outperforms *VLM-Vis* by achieving the lowest false positive rate and the highest F_1 score across all three business test datasets. Such result indicates that the incorporation of reinforcement learning enables the VLM to better acquire human-like UI comprehension and commonsense for display issue detection.

In general, the results of RQ1 indicate that each component of Minuku contributes to narrowing the gap between a general-purpose VLM and end users in terms of UI comprehension and display issue detection.

C. RQ2: Performance Comparing with Existing Methods

As shown in Table II, compared with three baselines, Minuku achieves the best F_1 while maintaining a FPR of 10%, demonstrating its superior capability in detecting display issues across five baselines.

Compared with *OwlEyes*, Minuku consistently outperforms it across all scenarios and evaluation metrics. This demonstrates that, under the same scale of training data, VLMs exhibit significantly stronger learning and generalization capabilities than traditional CNN-based detection approaches. Furthermore, compared to *M-YOLO*, even with three times more training data and performance superior to CNN-based methods, *M-YOLO* still falls short of Minuku in terms of recall and F_1. This highlights Minuku's ability to not only exceed traditional deep learning models in effectiveness, but also significantly reduce the scale of required training data.

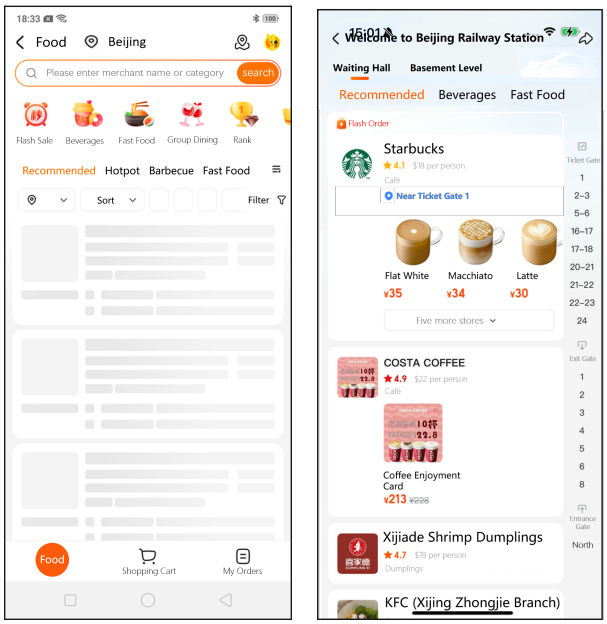
In addition, closed-source models exhibit either extremely low recall or high FPR, indicating a strong tendency to classify all screenshots as one class. This suggests that display issue detection remains a non-trivial task for general-purpose VLMs—even for a large, high-performing closed-source model. Moreover, Minuku outperforms all advanced closed-source models in F_1 score and *FPR*, despite having significantly fewer parameters, further demonstrating the cost-effectiveness of its fine-tuning procedure.

D. RQ3: Usability for Real-life Issues Detection

Given that Minuku and its baselines are trained on manually synthesized screenshots, their capability to detect real-world display issues remains unclear. Therefore, in RQ3, we evaluate their practical usability using historical display issues collected from practical commercial apps in Meituan. Considering that most closed-source models performed poorly in RQ2, we selected Gemini-2.5-Pro, the one with the highest F_1 score, as a representative for RQ3.

As shown in Table III, Minuku successfully recalled 20 out of 25 real-world display issues and outperformed all baselines in F_1 and *FPR*. Considering that the synthesized training anomalies inevitably differ from real-world issues, this result further highlights that Minuku has learned a generalizable human-like commonsense for UI expectation. Besides, although Minuku does not provide precise coordinates for the issues (a possible shortcoming compared to *OwlEyes* and *M-YOLO*), it manifests the lowest *FPR* among these methods and can provide a correct explanation for most of the recalls. Considering that Minuku is designed to assist UI testing experts rather than replace them, this limitation does not undermine its practical usability.

We acknowledge that the testing dataset used in RQ3 is rather small for a thorough validation of usability. Regretfully, due to the absence of a cost-effectiveness detection method,



(a) Blank issue on legacy devices. (b) Status bar occlusion in iOS 18.

Fig. 8: Examples of display issues detected in Meituan.

display issues have long received limited testing resources, even within a company that serves over 600 million users. Therefore, to further demonstrate the practical effectiveness of Minuku, we will present its real-world deployment process in Section V.

V. Case Studies

In this section, we show the real-life benefits by demonstrating the status of applying Minuku to display issues detection in Meituan. We summarize the lessons learned, which can shed light on the automation of UI testing.

For Minuku's deployment, it is now integrated as a service into Meituan's UI automated cloud device testing platform, enabling easy access for test engineers. When invoking Minuku, engineers typically provide a list of URL schemes [39] and select cloud devices to acquire screenshots online, or they can directly provide UI screenshots. Minuku then performs UI display issue detection, with the platform providing the result filtering and display of problematic UI screenshots along with their underlying reasons. Moreover, to facilitate local usage, a GGUF-quantized [40] model version is also provided, allowing 8-bit quantization to process screenshots on even laptops.

A. Compatibility Testing for Core Functionality

In fact, the display rendering of mobile apps is dependent on both UI implementation and its runtime environment. Given the typically massive functionality of commercial apps and the highly diverse user environments (*e.g.*, devices, screen sizes, display scales, and network conditions), UI testing typically prioritizes covering core functionalities under mainstream environments. This is particularly crucial for display issue detection, an area where automation has historically been limited.

Therefore, soon after Minuku's deployment, it is first leveraged to broaden the scope of existing display issues testing in Meituan. Specifically, starting from Meituan's core business of catering retail, we expanded the current display issue testing to cover legacy devices. Although legacy devices do not command a large user base, their earlier release, coupled with lower system versions and poorer computational capabilities, makes them inherently more susceptible to display issues compared to typical scenarios. Consequently, they served as the initial deployment scenario for Minuku.

Specifically, we involved 14 legacy devices (*e.g.*, Xiaomi Mi 8, OPPO V9, and Vivo S5) to process 4,336 URLs gathered from the catering retail business. Among the resulting screenshots, Minuku identified 260 potential UI display anomalies. Through manual analysis, 139 of these were attributed to anomalous backend data in the testing environment, such as being obscured by obvious test markers or rendering in blank pages, rather than actual display issues. From the remaining portion, we identified 30 screenshots containing clear display issues, as shown in Figure 8(a). The remaining 91 instances, which might have negligible subtle display issues, were considered false positives and discarded. Consequently, the actual false positive rate (FPR) was about 2.2%, and the false discovery rate (FDR) was 35%.

In summary, for compatibility testing, Minuku transformed the inspection of over 4,000 pages into the identification of just 260, reducing manual effort by more than 90%. This significantly boosted testing efficiency. Regarding cost, Minuku only requires deploying a 7B model. We estimate that even without a GPU, a Mac Mini M2 could process four thousand screenshots in approximately 48 hours. This demonstrates that Minuku not only improves existing display issue detection capabilities but also maintains a low testing cost, making it practical for large-scale and frequent testing in industrial settings.

B. Regression Testing under SDK Upgrades

Due to the massive user base of the iOS platform, ensuring timely adaptation to iOS updates has long been a critical concern for commercial app developers. Each year, when Apple releases a new version of iOS, its development tools, Xcode (as the integrated development environment, IDE) and the iOS SDK, are upgraded accordingly. Consequently, updating an existing app for submission to the App Store typically requires developers to repackage it using the latest SDK.

Such SDK upgrades often require extensive adaptation jobs, and improper handling of these changes can easily introduce display issues. Specifically, upgrades of SDK are often accompanied by internal changes to UI frameworks (*e.g.*, UIKit or SwiftUI). These changes may lead to display issues in native UI components (such as *UINavigationBar* or *UITabBar*) or result in visual occlusion with system elements like the notch area or the status bar. Moreover, SDK upgrades inevitably

introduce support for new device models and screen sizes, which often leads to changes in size-related controllers or layout behaviors. The lack of corresponding adaptation for UI layout code may lead to layout disruption. For large-scale commercial app providers, custom-designed UI libraries also require updates for compatibility with the new SDK in order to ensure consistent rendering across devices.

Therefore, we applied `Minuku` to the 2025 upgrade of Xcode 16 and iOS 18 [41], integrating it into the regression testing of four business modules in Meituan. `Minuku` examined 441 UI screenshots, captured from iPhone 13, 14, and 16 models, including both older and newer devices running iOS 18. Among the results, `Minuku` flagged 26 UI anomalies. Upon manual inspection by the authors, 5 of them were caused by abnormal data in the testing environment and thus did not qualify as display issues. A total of 10 reported issues were confirmed and fixed. As Figure 8(b) shows, the iOS SDK upgrade primarily introduced two problems: app components overlap with the status bar, and component loading failures (*i.e.*, presenting as blank regions). The remaining 11 cases were false positives, accounting for 2.6% of the inspected pages.

VI. Threats to validity

As `Minuku` is designed and implemented in Meituan, the first potential threat lies in `Minuku`'s generalizability to other apps or organizations. Although screenshots for evaluation are sourced from Meituan, the architecture and training process of `Minuku` are not specifically tailored to Meituan 's app, and it outperforms all baselines under the same training dataset. This suggests that our fine-tuning of a general-purpose VLM makes it possible to handle visual diversity with a small-scale dataset. Therefore, we believe that such an approach is appealing in industrial settings, where large-scale annotated datasets are often unavailable or prohibitively expensive to produce.

Another threat lies in our seemingly overly simplified characterization of display issues. In fact, display issues exhibit considerable diversity and could potentially be categorized into more than 30 subtypes [16]. However, considering that exhaustively discovering all display issue types is infeasible in industrial settings, a coarse-grained categorization is likely to enable the small-parameter VLM to better understand the task. We acknowledge that alternative or more fine-grained classification schemes may exist and could offer additional benefits. Nevertheless, this does not undermine the core contribution of our practical reduction of testing costs for display issues.

There's also a potential threat arise from the fact that `Minuku` is built upon an open-source VLM. As such, we acknowledge that its ability to detect UI display issues depends on the performance of the underlying base model. However, since Sections IV and Section V have demonstrated both `Minuku`'s performance improvement over existing detection techniques and its practical usability in industrial settings, we did not explore other base models in this work.

VII. Related Work

This section summarizes the relevant literature of `Minuku`, starting with a comprehensive review of existing display issue detection tools, followed by foundational research in semantic UI analysis methodologies.

A. Display Issue Detection

Despite slight variations in definition across different works [16], display issues generally refer to visual anomalies in user interfaces. Examples include overlapping components, truncated text, blank areas, or misaligned layout structures. In general, the detection of display issues can be approached from two directions: comparison-based methods and reference-free feature-driven approaches.

As for comparison-based methods, they identify display issues by comparing different sources of UI representations. Examples include contrasting the implemented UI with its design specification [42], or examining the UI inconsistency across different app versions [43] or execution settings [44]–[46]. However, for typically rapidly evolving commercial apps, UI differences are prevalent due to frequently added features and short development cycles. As a result, comparison-based methods tend to produce an unacceptably high rate of false positives in industrial practice.

As for feature-driven detection tools, existing academic attempts fall into two categories: heuristic-based methods that rely on manually crafted rules or layout constraints [20], [21], and learning-based methods that apply models like CNN to learn visual patterns from UI screenshots [1], [17], [19], [47]. Without external references, such methods rely on extracting the characteristics of display issues, recorded as heuristic rules or model parameters. Although also a learning-based approach, `Minuku` focuses on the underlying commonsense expectation for UI design rather than diverse characteristics of display issues. This reduces the reliance on large-scale training dataset and makes it more suitable for the detection of display issues under UI iteration.

B. Semantic UI Analysis

Semantic UI analysis is a foundational technology that supports automated App interaction [48]–[50] and testing [8], [51]. In terms of implementation, it can be divided into three categories: code-based, learning-based, and large model-based approaches. The code-based methods [4], [52] typically relied on UI source code to identify UI functionalities or page types. These methods are straightforward but suffer from issues like omitted visual information. Subsequently, deep learning-based methods [53]–[55] leverage visual or multi-modal neural networks (*e.g.*, CNN, YOLO, and ViT) for UI screenshot recognition, significantly enhancing image analysis capabilities. However, their reliance on large-scale, high-quality labeled data makes them usually costly to adopt in industrial settings.

Nowadays, the strong semantic comprehension of LLMs has made them highly advanced in the field of UI analysis. Existing tools often leverage LLMs to analyze view hierarchies [56], [57] or utilize visual-language models (VLMs) for screenshot

understanding [14], [58]. As for the adaptation of a specific UI analysis task, prompt engineering [8], [54] and fine-tuning [59], [60] are both widely adopted strategies. Inspired by these works, `Minuku` adopts a screenshot-based approach combined with lightweight fine-tuning, aiming to balance reasoning performance and computational cost for better industrial applicability.

VIII. Conclusion

This paper indicates that, due to the great diversity of UI designs, existing detectors generally exhibit high reliance on large volumes of training data in detecting display issues. Given the substantial manual efforts involved in providing such data, these detectors have long been considered cost-ineffective in industrial settings, leading to insufficient detection practices and a degraded user experience. To address this, we present `Minuku`, an industrial data-efficient UI display issue detector. By simulating human-like commonsense, `Minuku` is able to understand UI screenshots and reason potential display issues. Our experiments on customized benchmarks show that `Minuku` can effectively detect real-world display issues with limited training data, achieving 80% recall on historical display issues while maintaining a 2% false positive rate. Furthermore, practical industrial deployment on nearly 5,000 screenshots shows `Minuku`'s real-life usability in successfully detecting 40 previously unknown display issues with an unprecedentedly limited manual effort.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Project No. 62572127) and Meituan. Y. Zhou is the corresponding author.

References

[1] X. Liang, J. Qi, Y. Gao, C. Peng, and P. Yang, "AG3: automated game GUI text glitch detection based on computer vision," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 1879–1890.

[2] W. Liu, F. Lin, L. Guo, T. Chen, and A. E. Hassan, "Guiwatcher: Automatically detecting GUI lags by analyzing mobile application screencasts," *CoRR*, vol. abs/2502.04202, 2025.

[3] (Accessed: 2024) Ui/application exerciser monkey. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey

[4] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 245–256.

[5] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Z. Tian, Y. Huang, J. Hu, and Q. Wang, "Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 137:1–137:12.

[6] T. Su, L. Fan, S. Chen, Y. Liu, L. Xu, G. Pu, and Z. Su, "Why my app crashes? understanding and benchmarking framework-specific exceptions of android apps," *IEEE Trans. Software Eng.*, vol. 48, no. 4, pp. 1115–1137, 2022.

[7] Y. Huang, J. Wang, Z. Liu, Y. Wang, S. Wang, C. Chen, Y. Hu, and Q. Wang, "Crashtranslator: Automatically reproducing mobile application crashes directly from stack trace," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 18:1–18:13.

[8] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 100:1–100:13.

[9] S. Yu, C. Fang, M. Du, Y. Ling, Z. Chen, and Z. Su, "Practical non-intrusive GUI exploration testing with visual-based robotic arms," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 130:1–130:13.

[10] Y. Xiong, M. Xu, T. Su, J. Sun, J. Wang, H. Wen, G. Pu, J. He, and Z. Su, "An empirical study of functional bugs in android apps," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 1319–1331.

[11] Y. Xiong, T. Su, J. Wang, J. Sun, G. Pu, and Z. Su, "General and practical property-based testing for android apps," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 53–64.

[12] J. Wang, Y. Jiang, T. Su, S. Li, C. Xu, J. Lu, and Z. Su, "Detecting non-crashing functional bugs in android apps via deep-state differential analysis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 434–446.

[13] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, and Z. Su, "Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–31, 2021.

[14] Z. Liu, C. Li, C. Chen, J. Wang, M. Chen, B. Wu, Y. Wang, J. Hu, and Q. Wang, "Seeing is believing: Vision-driven non-crash functional bug detection for mobile apps," *CoRR*, vol. abs/2407.03037, 2024.

[15] Y. Hu, Y. Zhang, X. Wang, Y. Liu, S. Guo, C. Chen, X. Wang, and Y. Zhou, "Kuitest: Leveraging knowledge in the wild as gui testing oracle for mobile apps," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE Computer Society, 2025.

[16] L. Nie, K. S. Said, and M. Hu, "Sok: An exhaustive taxonomy of display issues for mobile applications," in *Proceedings of the 29th International Conference on Intelligent User Interfaces, IUI 2024, Greenville, SC, USA, March 18-21, 2024*. ACM, 2024, pp. 537–548.

[17] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Nighthawk: Fully automated localizing UI display issues via visual understanding," *IEEE Trans. Software Eng.*, vol. 49, no. 1, pp. 403–418, 2023.

[18] A. Saha, Y. Song, J. Mahmud, Y. Zhou, K. Moran, and O. Chaparro, "Toward the automated localization of buggy mobile app uis from bug descriptions," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 1249–1261.

[19] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Owl eyes: Spotting ui display issues via visual understanding," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 398–409.

[20] A. Alshayban and S. Malek, "Accessitext: automated detection of text accessibility issues in android apps," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 984–995.

[21] B. Yang, Z. Xing, X. Xia, C. Chen, D. Ye, and S. Li, "Don't do that! hunting down visual design smells in complex uis against design guidelines," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 761–772.

[22] Z. He, S. F. Huq, and S. Malek, "Enhancing web accessibility: Automated detection of issues with generative ai," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 2264–2287, 2025.

[23] P. Yu, J. Fei, H. Gao, X. Feng, Z. Xia, and C. H. Chang, "Unlocking the capabilities of large vision-language models for generalizable and explainable deepfake detection," *arXiv preprint arXiv:2503.14853*, 2025.

[24] Z. Li, X. Wu, H. Du, F. Liu, H. Nghiem, and G. Shi, "A survey of state of the art large vision language models: Alignment, benchmark, evaluations and challenges," *arXiv preprint arXiv:2501.02189*, 2025.

[25] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, M. Zhang, Y. K. Li, Y. Wu, and D. Guo, "Deepseekmath: Pushing the limits of mathematical reasoning in open language models," *CoRR*, vol. abs/2402.03300, 2024.

[26] C. Wang, A. Bochkovskiy, and H. M. Liao, "Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*. IEEE, 2023, pp. 7464–7475.

[27] (Accessed: 2025) Openvlm leaderboard. [Online]. Available: https://huggingface.co/spaces/opencompass/open_vlm_leaderboard

[28] (Accessed: 2024) Open llm leaderboard. [Online]. Available: https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard#/

[29] (Accessed: 2025) Qwen/qwen2.5-vl-7b-instruct. [Online]. Available: https://huggingface.co/Qwen/Qwen2.5-VL-7B-Instruct

[30] (Accessed: 2025) Cross-entropy. [Online]. Available: https://en.wikipedia.org/wiki/Cross-entropy

[31] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[32] S. Bai, K. Chen, X. Liu, J. Wang, W. Ge, S. Song, K. Dang, P. Wang, S. Wang, J. Tang, H. Zhong, Y. Zhu, M. Yang, Z. Li, J. Wan, P. Wang, W. Ding, Z. Fu, Y. Xu, J. Ye, X. Zhang, T. Xie, Z. Cheng, H. Zhang, Z. Yang, H. Xu, and J. Lin, "Qwen2.5-vl technical report," *CoRR*, vol. abs/2502.13923, 2025.

[33] (Accessed: 2025) Qwen/qwen3-8b. [Online]. Available: https://huggingface.co/Qwen/Qwen3-8B

[34] Q. Yu, Z. Zhang, R. Zhu, Y. Yuan, X. Zuo, Y. Yue, T. Fan, G. Liu, L. Liu, X. Liu, H. Lin, Z. Lin, B. Ma, G. Sheng, Y. Tong, C. Zhang, M. Zhang, W. Zhang, H. Zhu, J. Zhu, J. Chen, J. Chen, C. Wang, H. Yu, W. Dai, Y. Song, X. Wei, H. Zhou, J. Liu, W. Ma, Y. Zhang, L. Yan, M. Qiao, Y. Wu, and M. Wang, "DAPO: an open-source LLM reinforcement learning system at scale," *CoRR*, vol. abs/2503.14476, 2025.

[35] (Accessed: 2025) Model gpt-4.1. [Online]. Available: https://platform.openai.com/docs/models/gpt-4.1

[36] (Accessed: 2025) Claude 3.7 sonnet and claude code. [Online]. Available: https://www.anthropic.com/news/claude-3-7-sonnet

[37] (Accessed: 2025) Gemini 2.5 pro. [Online]. Available: https://deepmind.google/models/gemini/pro/

[38] (Accessed: 2025) Hello gpt-4o. [Online]. Available: https://openai.com/index/hello-gpt-4o/

[39] (Accessed: 2025) The app: Url scheme. [Online]. Available: https://www.w3.org/TR/app-uri/

[40] (Accessed: 2025) Gguf. [Online]. Available: https://huggingface.co/docs/hub/en/gguf

[41] (Accessed: 2025) Xcode 16 release notes. [Online]. Available: https://developer.apple.com/documentation/xcode-release-notes/xcode-16-release-notes

[42] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk, "Automated reporting of GUI design violations for mobile apps," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 165–175.

[43] K. Moran, C. Watson, J. Hoskins, G. Purnell, and D. Poshyvanyk, "Detecting and summarizing GUI changes in evolving mobile apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 543–553.

[44] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE Computer Society, 2017, pp. 308–318.

[45] Z. Tao, Y. Gao, J. Qi, C. Peng, Q. Wu, X. Chen, and P. Yang, "Neat: Mobile app layout similarity comparison based on graph convolutional networks," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, M. d'Amorim, Ed. ACM, 2024, pp. 104–114.

[46] Y. Su, C. Chen, J. Wang, Z. Liu, D. Wang, S. Li, and Q. Wang, "The metamorphosis: Automatic detection of scaling issues for mobile apps," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 101:1–101:12.

[47] Y. Su, Z. Liu, C. Chen, J. Wang, and Q. Wang, "Owleyes-online: a fully automated platform for detecting and localizing UI display issues," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 1500–1504.

[48] C. Zhang, Z. Yang, J. Liu, Y. Li, Y. Han, X. Chen, Z. Huang, B. Fu, and G. Yu, "Appagent: Multimodal agents as smartphone users," in *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems, CHI 2025, YokohamaJapan, 26 April 2025- 1 May 2025*, N. Yamashita, V. Evers, K. Yatani, S. X. Ding, B. Lee, M. Chetty, and P. O. T. Dugas, Eds. ACM, 2025, pp. 70:1–70:20.

[49] J. Wang, H. Xu, J. Ye, M. Yan, W. Shen, J. Zhang, F. Huang, and J. Sang, "Mobile-agent: Autonomous multi-modal mobile device agent with visual perception," *CoRR*, vol. abs/2401.16158, 2024.

[50] R. Niu, J. Li, S. Wang, Y. Fu, X. Hu, X. Leng, H. Kong, Y. Chang, and Q. Wang, "Screenagent: A vision language model-driven computer control agent," *CoRR*, vol. abs/2402.07945, 2024.

[51] Y. Hu, X. Wang, Y. Wang, Y. Zhang, S. Guo, C. Chen, X. Wang, and Y. Zhou, "Auitestagent: Automatic requirements oriented GUI function testing," *CoRR*, vol. abs/2407.09018, 2024.

[52] W. Li, Y. Jiang, C. Xu, Y. Liu, X. Ma, and J. Lu, "Characterizing and detecting inefficient image displaying issues in android apps," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, X. Wang, D. Lo, and E. Shihab, Eds. IEEE, 2019, pp. 355–365.

[53] Y. Hu, J. Gu, S. Hu, Y. Zhang, W. Tian, S. Guo, C. Chen, and Y. Zhou, "Appaction: Automatic GUI interaction for mobile apps via holistic widget perception," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 1786–1797.

[54] Y. Hu, H. Jin, X. Wang, J. Gu, S. Guo, C. Chen, X. Wang, and Y. Zhou, "Autoconsis: Automatic gui-driven data inconsistency detection of mobile apps," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 137–146.

[55] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable UI tests," in *Proc. of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. ACM, 2018, pp. 269–282.

[56] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "Fill in the blank: Context-aware automated text input generation for mobile GUI testing," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1355–1367.

[57] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 67:1–67:13.

[58] S. Cao, R. Chen, M. Pan, W. Yang, and X. Li, "Beyond manual modeling: Automating GUI model generation using design documents," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 91–103.

[59] Y. Qin, Y. Ye, J. Fang, H. Wang, S. Liang, S. Tian, J. Zhang, J. Li, Y. Li, S. Huang, W. Zhong, K. Li, J. Yang, Y. Miao, W. Lin, L. Liu, X. Jiang, Q. Ma, J. Li, X. Xiao, K. Cai, C. Li, Y. Zheng, C. Jin, C. Li, X. Zhou, M. Wang, H. Chen, Z. Li, H. Yang, H. Liu, F. Lin, T. Peng, X. Liu, and

G. Shi, "UI-TARS: pioneering automated GUI interaction with native agents," *CoRR*, vol. abs/2501.12326, 2025.

[60] K. You, H. Zhang, E. Schoop, F. Weers, A. Swearngin, J. Nichols, Y. Yang, and Z. Gan, "Ferret-ui: Grounded mobile UI understanding with multimodal llms," in *Computer Vision - ECCV 2024 - 18th European Conference, Milan, Italy, September 29-October 4, 2024, Proceedings, Part LXIV*, ser. Lecture Notes in Computer Science, A. Leonardis, E. Ricci, S. Roth, O. Russakovsky, T. Sattler, and G. Varol, Eds., vol. 15122. Springer, 2024, pp. 240–255.