An Empirical Study on UI Overlap in OpenHarmony Applications

Farong Liu, Mingyi Zhou*, Li Li*,
School of Software
Beihang University, Beijing, China

***Abstract*—UI overlap is a phenomenon where one UI component visually covers another. While this overlap is necessary to construct rich visual hierarchies, it is also a root cause of usability issues and performance bottlenecks. However, a systematic, data-driven understanding of its prevalence and patterns has been lacking. To bridge this gap, we conduct the first large-scale empirical study on UI overlap in the OpenHarmony ecosystem. We analyze 100 popular apps, classifying 33,262,624 overlap instances through a novel three-tiered taxonomy. Our findings reveal that *high-cost occlusion* is a critical and previously hard-to-detect performance defect where resource-intensive components are rendered while visually obscured. We propose HCO-Eye, an innovative tool that leverages multimodal vision-language models (VLMs) to automatically detect such issues, successfully identifying 34 high-cost occlusion cases in commercial apps. Our study not only provides the first comprehensive understanding of UI overlap in OpenHarmony but also offers a practical tool to automatically diagnose complex performance-related UI bugs. Our tools are publicly available.**

I. Introduction

The Graphical User Interface (GUI) serves as the visual bridge connecting users and software applications. With the rise of the OpenHarmony ecosystem [1], [2] and the popularization of declarative UI frameworks like ArkUI, the development paradigm for mobile applications is undergoing a profound transformation, leading to a significant increase in the complexity of UI hierarchies and interaction patterns. As user expectations for richer, more interactive, and visually dynamic experiences continue to grow, developers are increasingly adopting complex design patterns. This shift is driven by the need to deliver sophisticated functionality and dense information within the constrained screen space of mobile devices.

UI overlap [3], [4] is an indispensable ”double-edged sword” in modern UI design. On one hand, it is a necessary design technique for implementing modern features such as floating action buttons, stacked cards, and modal dialogs, which are essential for creating rich visual hierarchies and fluid user experiences. On the other hand, it is a major source of problems, leading to usability issues like obscured text and unclickable buttons, as well as performance bottlenecks caused by phenomena such as overdraw and occlusion. Effectively managing and leveraging UI overlap has thus become a core challenge in ensuring application quality and user experience.

*Corresponding authors

Nevertheless, the characterization and detection of UI overlap issues are non-trivial. First, little is known about how prevalent overlap is in the OpenHarmony ecosystem and what the common patterns are. Second, existing UI defect detection tools are often platform-specific, typically targeting major ecosystems like Android and iOS, and cannot be directly applied to OpenHarmony. Third, and most importantly, traditional static [5], [6] or dynamic [7] analysis tools lack the semantic understanding to distinguish between intentional design and unintentional defects, let alone identify complex performance issues like high-cost occlusion. This paper defines high-cost occlusion as a scenario where a high-cost component is rendered despite being fully or partially covered by other UI elements, leading to wasted computational resources.

For the purpose of this study, we define a *high-cost component* based on the following criteria:

- **Heavy Rendering Type:** Components that are inherently expensive to render, such as a Video Player, Complex Animation, Map, Camera Preview, Long Image List, or WebView.
- **Heavy Computation/IO Type:** Real-time refreshing components that involve significant background processing, such as Complex Charts or Data-Heavy Lists that update frequently.

To bridge these gaps, this paper conducts the first large-scale empirical study of UI overlap. We choose the OpenHarmony platform as the target. OpenHarmony is a new mobile platform with rapidly evolving applications, which usually have a complex UI structure. This study is guided by the following three core Research Questions (RQs):

- **RQ1: The Prevalence of UI Overlap**. How prevalent is UI overlap in real-world OpenHarmony applications?
- **RQ2: The Classification of UI Overlap**. What are the common, reproducible patterns of UI overlap?
- **RQ3: A Detection Tool for Performance-Related Overlap**. To what extent can we automatically identify performance-related overlap issues?

To answer these questions, we perform an in-depth analysis of 100 popular apps, propose a novel taxonomy to classify overlap patterns, and design an innovative tool, HCO-Eye, which leverages a multimodal vision-language model to automatically detect performance-critical defects. The main contributions of this paper include:

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00274

- An empirical study on 100 real-world OpenHarmony apps to understand the prevalence and patterns of UI overlap.
- A novel three-tiered taxonomy for UI overlap, which systematically classifies instances based on their structural, existential, and scenario-based properties.
- An innovative tool, HCO-Eye, and a corresponding benchmark, HCOBench, demonstrating the feasibility of using VLM to detect dynamic, performance-related UI defects.
- We open-source our dataset and tool at: https://github.com/SMAT-Lab/HapOverlap.

II. Background

A. OpenHarmony ArkUI Framework

This section introduces the foundational concepts of the `OpenHarmony ArkUI` framework [8], with a specific focus on the layout mechanisms that enable `UI overlap`.

1) Declarative UI Paradigm: This study is situated within the OpenHarmony ecosystem, focusing on its native UI framework, `ArkUI`. `ArkUI` is a modern, declarative framework designed for building user interfaces across a wide range of devices. It utilizes `ArkTS`, an extension of TypeScript, to enable developers to describe the UI's state and appearance in a concise and intuitive manner. In this paradigm, the UI is a function of the application's state; when the state changes (e.g., via variables decorated with `@State`), the framework automatically and efficiently updates the relevant parts of the UI, abstracting away the complexities of manual view manipulation.

2) Box Model: The core layout philosophy of `ArkUI` is based on a hierarchical structure of `components`. Each `component` occupies a region governed by a box model, which is fundamental to its sizing and positioning, as illustrated in Figure 1. This model is composed of several nested layers:

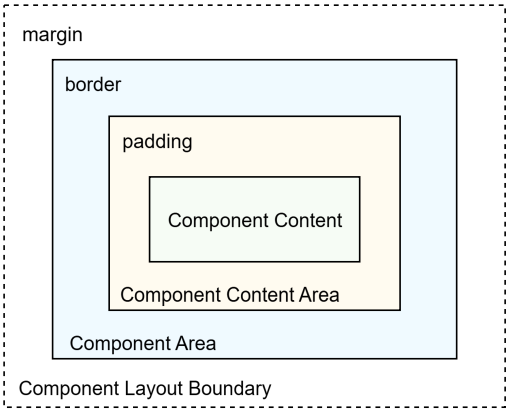

Fig. 1: The Box Model of an ArkUI Component.

This model dictates the layout process: a parent container passes size constraints to its children's `Component Area`, and when arranging them, it considers the full space occupied by their `Component Layout Boundary`. Based on this principle, `ArkUI` provides a comprehensive set of layout containers to manage component arrangement. Foundational containers such as `Row` and `Column` arrange components linearly along a single axis without overlap, serving as the basis for most common, non-overlapping UI structures.

3) Containers and Layouts: To construct more complex, multi-layered interfaces, which are central to the phenomenon of `UI overlap`, `ArkUI` provides specialized containers and attributes. The most direct and fundamental of these is the `Stack` container. By design, `Stack` arranges its child components along the Z-axis, placing each subsequent child on top of the previous one. This inherent stacking behavior is the primary intended mechanism for creating UI overlap in OpenHarmony applications. The stacking order can be explicitly managed using the `zIndex` attribute. A child component with a higher `zIndex` value will be rendered on top of its siblings with lower values, regardless of their declaration order.

Beyond the `Stack` container, UI overlap can also manifest through other layout mechanisms. The `RelativeContainer` allows for complex, non-linear arrangements where a child's position is defined relative to its siblings or the parent container using `alignRules`, which can explicitly position one component on top of another. Additionally, universal attributes such as `position` (for absolute positioning within a parent) and `offset` (for relative positioning from an element's original location) can programmatically shift components, potentially causing them to overlap with others in the UI hierarchy. For the purpose of this study, understanding these mechanisms, particularly the `Stack` container, is fundamental to identifying, analyzing, and classifying instances of UI overlap.

B. Multimodal Large Models in Software Engineering

In recent years, the field of software engineering has witnessed significant advancements with the emergence of VLMs [9]. From pioneering models represented by GPT-4V [10] to subsequent models like ChatGPT-4o [11], Gemini [12], and Qwen-VL [13] with enhanced image reasoning capabilities, these technologies have substantially improved the understanding of mobile GUIs. A key breakthrough in this area is the ability of VLMs to understand and operate on UIs directly from pixels, a capability that has been shown to surpass methods reliant on structured information like HTML, as demonstrated by the success of works such as CogAgent [14]. This trend has also spurred the development of specialized agent models designed for GUI interaction, such as UI-TARS [15] and SeeClick [16], along with corresponding evaluation benchmarks [17], [18], [19], [20]. These research achievements showcase the immense potential of VLMs in the deep semantic understanding of UIs. This provides a solid theoretical basis and technical feasibility for our adoption of VLM technology in RQ3 to address more challenging UI performance issues that require such deep semantic comprehension.

III. Motivation

We use a motivating example to illustrate what high-cost occlusion is, why it leads to significant performance waste, and why it is challenging for traditional analysis tools to detect.

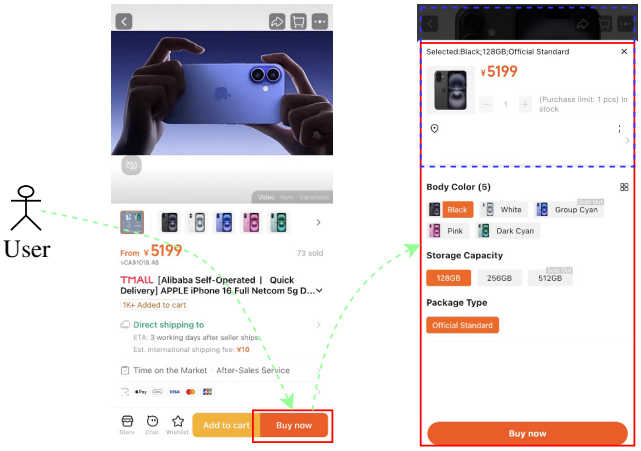

Fig. 2: A Motivating Example of High-Cost Occlusion in an E-commerce App

Consider the modern e-commerce application shown in Figure 2. At the top of its product detail page, a promotional video showcases the item's features, while a prominent "Buy Now" button sits below. When a user clicks this button, a bottom sheet slides up, presenting options for product specifications (e.g., size, color), payment methods, and order notes. This is a very common and intuitive checkout flow. However, from a performance perspective, it can hide a silent disaster.

As the user hesitates and spends time making choices on the bottom sheet, the promotional video in the background, now completely occluded, may continue to play and decode. At this moment, the GPU is still working hard to render unseen video frames, and the decoder continues to consume significant CPU and battery power, all of which are completely meaningless to the user and a pure waste of system resources [21].

Traditional performance analysis tools, such as Profilers [22] and GPU Inspector [23], might detect an increased GPU or CPU load during this checkout process, or even a drop in the overall frame rate [24], [25], but they cannot correlate this performance issue with the UI event of "the product video being occluded by the checkout panel." They lack the ability to understand the visual semantics of this operation. They do not comprehend that the "bottom sheet" is a temporary overlay, nor that the "video" is a high-cost component. This "semantic blindness" is the Achilles' heel of traditional tools and serves as the core motivation for our design of HCO-Eye to address such problems.

IV. Study Design

A. Experiment Environment

All experiments are conducted on a HUAWEI Mate60 device running OpenHarmony 5.1 to ensure consistency.

B. Dataset

Our study relies on a curated dataset named HarmonyAppSet100. Sourced from the official Huawei AppGallery, we selected 100 unique, popular native OpenHarmony applications based on rigorous criteria: over 1,000,000 downloads, an update within the last year, and successful installation on our test device. This dataset covers a wide range of categories, forming a representative sample of the ecosystem.

C. Research Procedure Overview

Our research procedure is structured in three phases, one for each research question, as illustrated in Figure 3. For RQ1, we quantitatively analyze overlap prevalence using H-Analyzer on the HarmonyAppSet100. For RQ2, we manually annotate a 1,000-instance sample to build a qualitative classification taxonomy. Finally, for RQ3, we design and implement the HCO-Eye tool and evaluate its performance on our newly constructed HCOBench benchmark.

This following sections present the results of our empirical study, organized by each research question.

V. RQ1: Prevalence of UI Overlap

We employ H-Analyzer to systematically crawl each app in the HarmonyAppSet100 for a duration of 30 minutes. During the exploration, the tool continuously analyzes the UI state. If an overlap is detected, it archives the current screenshot along with its associated UI component tree. The resulting collection of all detected overlap instances forms a comprehensive dataset, allowing us to conduct our analysis at three distinct granularities:

- **Application-level:** The percentage of apps with at least one overlap.
- **Screen-level:** The percentage of screens with overlaps and the density of overlaps per screen.
- **Instance-level:** The total count, structural relationships, area ratios, and involved component types.

A. Automated Analysis Tool

To facilitate automated analysis, we develop a specialized tool named H-Analyzer. The tool's architecture consists of three main modules: a UI crawler, an overlap detector, and a scene archiver. First, the UI crawler navigates through application screens. It leverages the OpenHarmony dynamic test framework, HapTest [7], and employs a random exploration strategy. For each visited screen, the crawler invokes the overlap detector module. Next, the overlap detector analyzes the UI component tree to identify overlapping elements. It first prunes the component pairs to eliminate duplicates. Then, it calculates the overlap area for the remaining pairs based on their bounding boxes and hierarchical layers. Finally, if the detector finds any overlap, the scene archiver saves the current context. This captured scene includes the current screenshot, the complete UI component tree, and related contextual information for subsequent analysis.

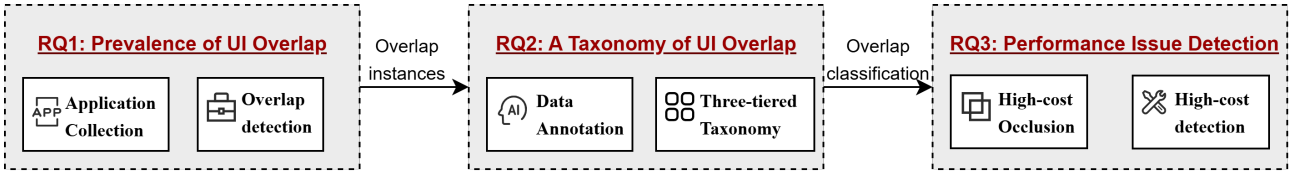

Fig. 3: An overview of our empirical study on UI Overlap in OpenHarmony Applications.

B. Results and Analysis

1) Application-level: At the application level, a remarkable 100 out of 100 tested apps (100%) contain at least one UI overlap instance in their explored screens. For a more granular analysis of this prevalence, we first classify the 100 apps into eight distinct categories. To highlight the most severe cases within each domain, our analysis then calculates the average from the top five most affected apps. As shown in Figure 4, which plots these averages, our analysis reveals that `Media` and `Shopping` apps exhibit the most significant overlap, with averages of 70.6 and 67.8 screens, respectively.

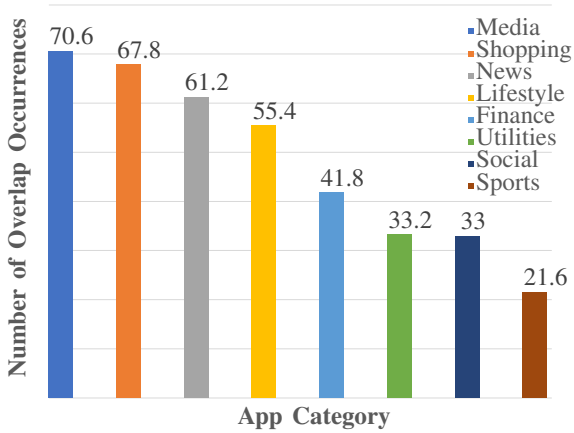

Fig. 4: Comparison of Overlap Occurrences Across App Categories.

2) Screen-level: At the screen level, the high frequency of overlap persists. Our analysis of the 100 apps identifies a total of 13,610 screens with at least one overlap instance. This figure represents a substantial 36.8% of the 37,017 unique screens explored. This finding demonstrates that this is not a niche issue but a widespread characteristic of modern UI design. On these screens, the mean number of overlaps is 9.02, with a median of 5. The histogram in Figure 5 illustrates a long-tail distribution: while most screens feature a manageable 1-10 overlaps, a small number of "hotspot" screens contain more than 50.

3) Instance-level: At the instance level, our analysis covers 33,262,624 individual UI overlap instances. Our first and most striking finding concerns their structural relationships. We find the vast majority (99.6%) of these instances occur between components with an `Ancestor-Descendant` relationship. Conversely, overlaps between `Sibling` components are exceedingly rare (0.4%; 122,857 instances). This result shows

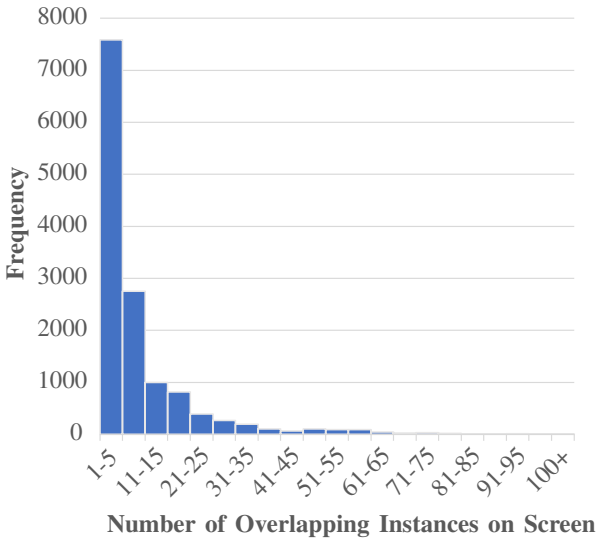

Fig. 5: The distribution of the number of overlapping instances on the screen.

that UI overlap in OpenHarmony is almost exclusively a phenomenon of hierarchical composition.

To further characterize these instances, we analyze the overlap area ratios. The Empirical Cumulative Distribution Function (ECDF) in Figure 6 shows a curve closely aligned with the `y=x` diagonal, which suggests that the overlap ratios approximately follow a uniform distribution.

Table I presents the top 20 most frequent component overlap pairs, reveals that overlaps are dominated by foundational layout components. The `genericContainer|genericContainer` pair is overwhelmingly prevalent, highlighting its central role in layering. This trend is reinforced by frequent self-overlaps of `Stack`, `Column`, and `Row`, indicating that stacking similar elements is a common design pattern. While our complete dataset contains 1,220 distinct overlapping pair types, a clear long-tail effect is visible, with most types occurring infrequently.

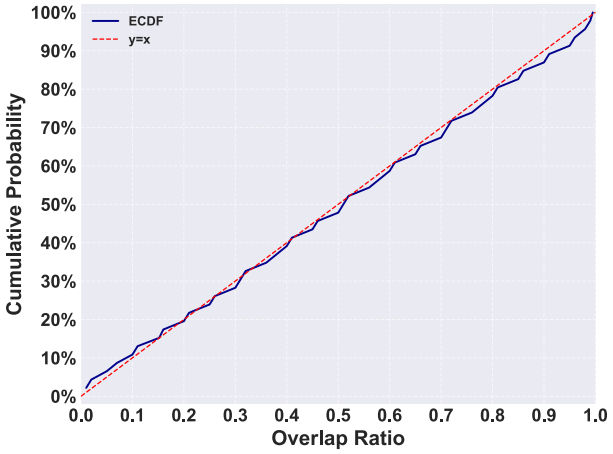

Fig. 6: ECDF of overlap ratios, indicating a near-uniform distribution.

TABLE I: Top 20 Most Frequent UI Component Overlap Pairs and Their Counts

Component Pair	Count	Percentage
genericContainer|genericContainer	28489	23.1887%
Row|genericContainer	4463	3.6327%
image|genericContainer	3534	2.8765%
Custom|Custom	3237	2.6348%
Stack|Stack	2753	2.2408%
__Common__|__Common__	1888	1.5367%
__Common__|Column	1730	1.4081%
genericContainer|staticText	1693	1.3780%
staticText|genericContainer	1660	1.3512%
Column|Column	1594	1.2974%
__Common__|Stack	1561	1.2706%
Stack|Column	1555	1.2657%
Stack|Dialog	1486	1.2095%
genericContainer|image	1385	1.1273%
Row|Row	1314	1.0695%
genericContainer|Column	1288	1.0484%
Stack|__Common__	1271	1.0345%
CustomFrameNode|CustomFrameNode	1222	0.9947%
genericContainer|__Common__	1098	0.8937%
Column|__Common__	1088	0.8856%
Top 20 Total	**64309**	**52.3446%**

Answer for RQ1: UI overlap is an exceptionally prevalent and fundamental design paradigm in OpenHarmony applications, consistently observed across all analyzed apps, a significant portion of screens, and overwhelmingly implemented through hierarchical component layering.

VI. RQ2: CLASSIFICATION OF UI OVERLAP

Having established the prevalence of UI overlap, RQ2 investigates its common patterns. To do this, we perform a qualitative analysis on a representative subset of 1,000 `Sibling` overlap instances detected in RQ1. To ensure a diverse representation of applications, these samples are meticulously collected by systematically sampling an equal number of instances from each app in our dataset. Two researchers independently annotated each sample by examining its screenshot, component tree, and dynamic context. Any disagreements in their annotations are subsequently resolved through discussion to reach a final consensus. This analysis leads us to inductively derive a three-tiered taxonomy for UI overlap.

A. A three-tiered Taxonomy.

As illustrated in Figure 7, we design a three-tiered taxonomy to build a systematic and objective classification framework. This taxonomy deconstructs each overlap instance along three orthogonal dimensions: its structural formation, temporal origin, and functional purpose.

- **Tier 1: Structural Root:** Classifies an overlap based on the components' relationship in the view hierarchy, categorized as either `Ancestor-Descendant` or `Sibling`.
- **Tier 2: Existential Root:** Classifies the cause of the overlap's existence over time, categorized as either `Static Layout Design` or `Dynamic Event-Driven`.
- **Tier 3: Meta-Scenario:** Classifies the functional purpose of the overlap into one of six mutually exclusive scenarios, from `MS1: Basic Layout` to `MS6: Layout Defect`. This tier also helps explain how defects like "Content Overflow" or "High-Cost Occlusion" can co-occur with benign scenarios.

B. Results and Analysis

Our analysis of the 1,000 sampled sibling overlaps reveals distinct patterns in how and why non-hierarchical UI components overlap in OpenHarmony applications. We present our findings in two parts: first, an analysis of the distribution and purpose of these overlaps, and second, a deeper dive into their underlying design archetypes.

1) Distribution of Overlaps: The initial breakdown shows that `Static Layout Design` is the predominant source (60.7%), compared to overlaps from `Dynamic Event-Driven` causes (39.3%). This suggests most non-hierarchical overlaps are built into the UI's fundamental structure. Figure 8 provides a detailed breakdown of how these instances are distributed across the functional meta-scenarios.

Static overlaps exhibit a clear dichotomy, serving two contrasting purposes: intentional design and unintentional defects. The most frequent meta-scenario is `MS1: Basic Layout & Background`, which constitutes 44.2% of static instances. This signifies that developers frequently and deliberately use sibling overlaps as a foundational technique to construct the core visual architecture of an interface. Following this, `MS2: Information & Visual Enhancement` accounts for 27.2%, reinforcing that the static placement of decorative or informational elements over adjacent sibling components is a popular and intentional design choice. Critically, however, `MS6: Layout Defect & Adaptation Failure` is the third most common cause, at 23.7%. This is a significant finding, implying that nearly a quarter of all static sibling overlaps are unintentional bugs resulting from

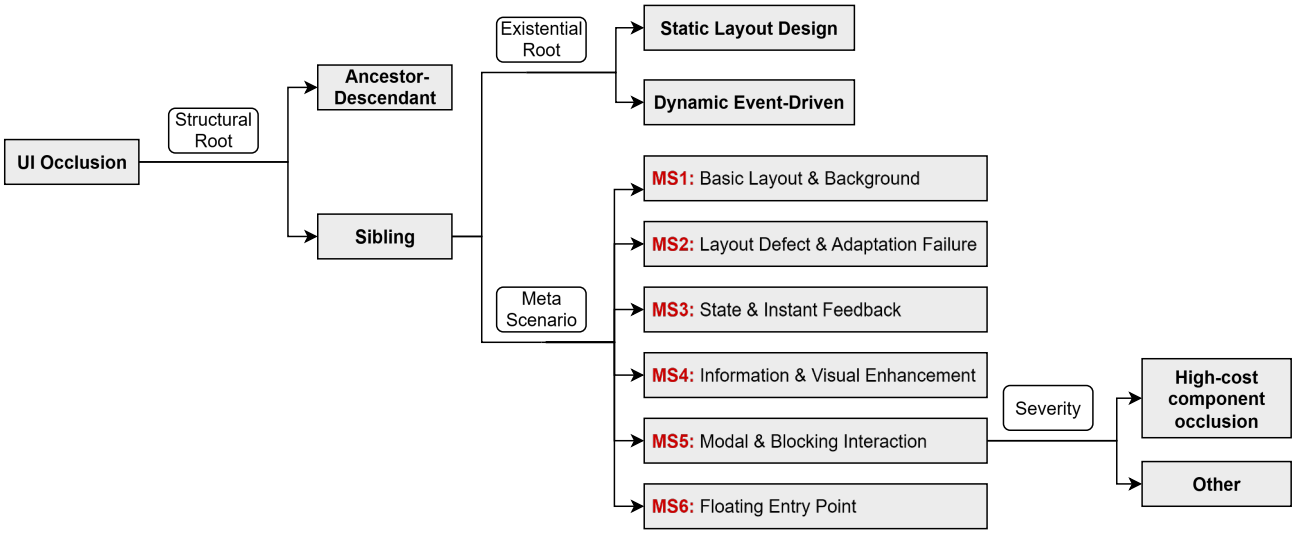

Fig. 7: The three-tiered taxonomy for classifying UI overlap.

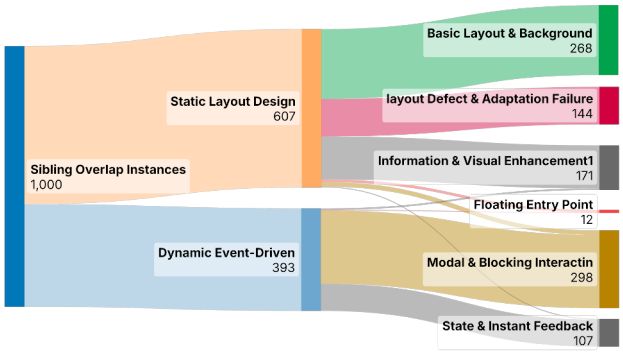

Fig. 8: Classification results for sampled overlaps.

implementation errors such as content overflow or adaptation issues.

For dynamically triggered overlaps, the purpose is overwhelmingly centered on temporary, flow-interrupting interactions. `MS5: Modal & Blocking Interaction` is the dominant cause by a large margin, at 71.5% of all dynamic instances. This strongly suggests that when sibling components overlap dynamically, it is most often because a modal element—such as a pop-up, menu, or dialog—appears and temporarily covers other sibling components on the screen. The second most common reason is `MS3: State & Instant Feedback` (26.5%), which includes temporary feedback like Toasts or loading indicators that overlay adjacent content.

2) Design Archetypes of Overlap: A deeper analysis reveals a strong correlation between an overlap's classification and its underlying design archetype. Table II summarizes the key patterns observed for each combination of root cause and meta-scenario.

The patterns observed for `Static (A)` overlaps define the persistent, visible framework and ornamentation of the UI. For instance, `A-MS1` is characterized by foundational patterns like the use of fixed navigation bars and the structural stacking of generic containers, which together form the UI's "skeleton." Similarly, `A-MS2` manifests as integral decorative patterns, such as badges on icons and promotional overlays on images, that are built directly into the static layout. The patterns for `A-MS4`, like fixed Floating Action Buttons, are also permanent fixtures of the interface. Even when overlaps are unintentional, as in `A-MS6`, their patterns, such as content overflow and incorrect spacing, are errors rooted in the initial, static layout declaration.

In stark contrast, the patterns associated with `Dynamic (B)` overlaps represent the application's responsive behavior to runtime events. The most common dynamic patterns fall under `B-MS5`, which includes a wide variety of ephemeral, flow-interrupting implementations like user-initiated action sheets, system-triggered alerts, and authentication flows. Patterns for `B-MS3` are similarly transient, exemplified by temporary Toasts, loading spinners, and contextual input aids that appear in direct response to user actions or system state changes. The patterns for `B-MS2`, such as dynamic notification badges and new content banners, serve to convey real-time information. These dynamic patterns are not part of the UI's static architecture; instead, they temporarily alter the visual hierarchy to manage runtime events and user interactions.

TABLE II: Summary of UI Overlap Patterns by Root Cause and Meta-Scenario

Root Cause	Meta-Scenario	Description	Key Pattern(s)
Static (A)	**MS1**	Provides UI elements' basic containers and visual backgrounds, forming the UI "skeleton."	Fixed UI Element Overlays (Nav/Toolbars, Search Bars); Generic Container Stacking for Structural Layout; Content Elements Overlapping Backgrounds or Containers; Page Sectional Composition with Implicit Overlap; Specific Control or Feature-Related Overlays (e.g., map markers).
	MS2	Attaches additional information or decoration onto existing UI elements.	Badges and Status Indicators on icons/avatars; Promotional and Pricing Overlays on media; Media Information and Interaction Overlays; Contextual Toolbars and Floating Action Elements; Integrative Visual and Textual Annotations.
	MS3	Temporarily reflects component state changes or user operation feedback.	Centralized Loading Animations covering content.
	MS4	Permanent function entry points detached from the main content flow.	Fixed Bottom Navigation/Toolbars; Floating Action Buttons for key actions; Persistent Functional Strips/Bars (Non-Navigation); Floating Promotional/Advertisement Entry Points; Mini/Persistent Media Players.
	MS5	Interrupts main process, forces user to complete tasks in a sub-environment.	Full-Screen Blocking Views (Authentication, Onboarding); Contextual Modal Dialogs/Action Sheets (User Input/Choices); Input-Activated Suggestion/Selection Panels; Feature-Specific Blocking Interfaces (e.g., content pickers).
	MS6	Unexpected, non-intentional overlaps caused by engineering implementation issues.	Content/Text Overflow; Incorrect Spacing or Alignment; Fixed/Sticky Element Misalignment; Unhandled Empty States or Loading Placeholders; General Unintended Component Overlaps.
Dynamic (B)	**MS1**	None.	*(No prominent patterns observed in our sample.)*
	MS2	Attaches additional information or decoration onto existing UI elements, triggered dynamically.	Dynamic Notification Badges & New Content Banners; User-Activated Contextual Overlays/Controls; Real-time Media Enhancement Overlays.
	MS3	Temporarily reflects component state changes or user operation feedback, triggered dynamically.	Transient Textual/Visual Alerts (Toasts & brief Pop-ups); Loading, Error, & Empty State Overlays; Contextual Interactive Controls & Input Aids; Event-Triggered Information/Action Panels; Real-time Media Stream Enhancements & Feedback.
	MS4	Permanent function entry points, appearing dynamically.	Dynamic Mini/Persistent Media Player Controls; Floating Customer Service/Support Avatars & Icons.
	MS5	Interrupts main process for tasks, appearing dynamically.	User-Initiated Selection & Action Sheets; Authentication and Transaction Flows; Contextual Menus and Input Aids; System or App-Triggered Alerts and Confirmations; Full-Screen Content or Feature Overlays.
	MS6	None.	*(No prominent patterns observed in our sample.)*

Answer for RQ2: The common, reproducible patterns of UI overlap can be systematically classified by a two-tiered taxonomy that first distinguishes their root cause as either `Static Layout Design` or `Dynamic Event-Driven`, and then categorizes their functional purpose into one of six mutually exclusive meta-scenarios, most notably `Basic Layout & Background`, `Modal & Blocking Interaction`, and `Layout Defect`.

VII. RQ3: A DETECTION TOOL FOR PERFORMANCE-RELATED OVERLAP

Our first two research questions establish the prevalence and systematically classified the patterns of UI overlap. A key finding from our classification in RQ2, particularly within the `B-MS5` category, is the identification of high-cost occlusion as a significant, yet hard to detect performance defect. As discussed in our Motivation section, this type of overlap can lead to significant performance waste. This section details

our approach, which involves building a new benchmark and designing an innovative, AI-driven tool to tackle this problem.

A. Research Goal and Challenges

The explicit goal of this research is to automatically detect high-cost occlusion defects. They occur when a dynamic modal element covers a resource-intensive background component that continues to render unnecessarily. The primary challenge is that detection requires semantic understanding, not just geometric analysis. Traditional methods fail because they cannot comprehend the context. For instance, they cannot recognize an occluded component as a high-cost Video or correlate the occlusion with a Dialog event. Our approach must interpret visual content and structural logic simultaneously to identify this specific, costly interaction.

B. Benchmark Construction: HCOBench

To enable rigorous evaluation, we construct HCOBench, a focused benchmark for high-cost occlusion. The process starts with 298 initial candidates drawn from the `B-MS5` category. We then curate a final set of 152 challenging samples, which includes every instance identified as positive by at least one VLM or a human annotator. Through final manual verification of this set, we confirm that the benchmark contains 38 true positive cases.

C. Tool Design: HCO-Eye

To address these challenges, we design and implement HCO-Eye, a novel tool that leverages multimodal large models to detect high-cost occlusion. The core idea is to simulate the diagnostic process of a human performance expert. As illustrated in Figure 9, the architecture of HCO-Eye follows a two-stage inference pipeline.

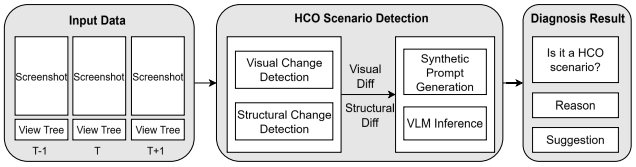

Fig. 9: HCO-Eye: A Two-Stage Automated Tool for High-Cost Occlusion Detection and Diagnosis.

1) Stage 1: Change Detection and Description: This stage analyzes a sequence of consecutive screenshots and their corresponding component trees to understand the UI's dynamic changes.

- **Visual Change Detection:** To capture visual changes, HCO-Eye provides two consecutive screenshots to a VLM. It uses a specific prompt (`PROMPT_DIFF_DESCRIPTION`)[1] to generate a concise, natural language description of the visual diff. For example, it might output: "A dialog box appeared at the center of the screen, covering the background content."
- **Structural Change Detection:** Concurrently, HCO-Eye compares the component trees of the two states to identify newly added components. This process generates a structural diff, which precisely describes which components have appeared and where they are located on the screen.

2) Stage 2: Performance Opportunity Inference: This stage acts as the core diagnostic engine. It takes the visual change description, the structural change description, and the current screenshot as a comprehensive input package. It then queries the VLM with a carefully designed prompt (`PROMPT_INSTRUCTIONS`)[2] that instructs the model to perform an expert-level chain-of-thought reasoning. Finally, the VLM generates the diagnosis result, which contains three key fields:

- **Label:** A binary classification ('Yes'/'No') indicating if a high-cost occlusion scenario is detected.
- **Reason:** A natural language explanation detailing the rationale behind the classification.
- **Suggestion:** An actionable recommendation for developers to fix the potential performance issue.

D. Results and Analysis

We evaluate HCO-Eye against the 152 cases in HCOBench using standard precision, recall, and F1-score metrics. The results, presented in Table III, demonstrate the high efficacy of our method, HCO-Eye, particularly when powered by the Gemini model. Our approach achieves an overall accuracy of 96.71% and an F1-score of 93.12%. This strong performance is underpinned by a very high precision of 97.14%, indicating that an issue flagged by HCO-Eye is almost certainly a genuine defect (34 true positives out of 35 flagged issues). Furthermore, with a recall of 89.47% (correctly identifying 34 out of the 38 total defects), the tool proves its robust capability to detect the vast majority of high-cost occlusion problems.

The ablation study provides critical insights into the synergy between the different components of our method. The most significant finding is that both textual and visual difference signals are crucial for achieving optimal performance, with visual changes being the primary driver.

(1) Contribution of Visual Difference: The *Visual Difference* is a textual description of content changes between frames generated by a VLM. This component is fundamentally important.

[1] e.g., "Describe the visual changes between the first image and the second image. Focus on new elements appearing, existing elements disappearing, or significant movements/changes of elements. Be concise and objective."

[2] The prompt is structured into several key sections to ensure a robust and consistent analysis:

- **Persona Setup:** Defines the VLM's role as a "top-tier mobile application performance optimization expert."
- **Key Definitions:** Provides precise criteria for what constitutes a "High-Cost Component" (e.g., video players, complex animations, maps) and "Significantly Obscured" (e.g., 80% of the component's area is covered by an opaque element).
- **Analysis Process:** Mandates a step-by-step reasoning chain: first identify high-cost components based on the definitions, then check for significant occlusion.
- **Error Handling:** Instructs the model to ignore common visual artifacts, such as content being abruptly cut off at image edges, clarifying that these are screenshot tool issues and not genuine occlusion events.
- **Output Requirements:** Specifies the exact output format, typically a JSON object containing the *label*, *reason*, and *suggestion* fields.

TABLE III: Ablation study of HCO-Eye components

Model	Setting	Performance Metrics (%)			
		Acc	Prec	Rec	F1
Llama[a]	HCO-Eye (Ours)	63.16	40.00	94.74	56.25
	w/o Structural Diff	**69.74**	**45.24**	**100.00**	**62.30**
	w/o Visual Diff	53.29	31.46	73.68	44.09
	w/o Structural and Visual Diff	50.00	30.21	76.32	43.28
OpenAI[b]	HCO-Eye (Ours)	**76.97**	**52.63**	78.95	**63.16**
	w/o Structural Diff	70.39	44.93	**81.58**	57.94
	w/o Visual Diff	73.03	44.44	31.58	36.92
	w/o Structural and Visual Diff	74.34	47.83	28.95	36.07
Gemini[c]	HCO-Eye (Ours)	**96.71**	**97.14**	**89.47**	**93.12**
	w/o Structural Diff	95.39	96.97	84.12	90.14
	w/o Visual Diff	86.18	77.42	63.16	69.57
	w/o Structural and Visual Diff	84.21	73.33	57.89	64.71

[a] meta-llama/llama-4-maverick. [b] openai/gpt-4o-mini. [c] gemini-2.5-flash.

Removing this component from the Gemini-based tool causes the F1-score to plummet by over 23 points (from 93.12% to 69.57%) and recall to drop by 26 points. This dramatic decrease highlights a core principle: structural information alone is insufficient. The VLM's ability to "see" and describe visual changes allows it to understand the semantic nature of the occluded content. For instance, it can differentiate between a low-cost static image being covered versus a high-cost, continuously rendering video player being hidden. This semantic context, which is entirely absent in component trees, is the primary reason for the large performance gain.

(2) Contribution of Textual Difference: The *Textual Difference*, which captures changes in the UI's component tree structure, acts as a vital complementary signal. While less impactful than visual analysis, its inclusion consistently enhances performance on capable models like Gemini, boosting the F1-score by approximately 3 points (from 90.14% to 93.12%). This improvement stems from its ability to provide precise, unambiguous cues about state changes. For example, the appearance of a `Dialog` or `Modal` component in the tree is a strong, explicit indicator of a potential occlusion event. This structural data helps to ground the VLM's reasoning, reducing ambiguity and preventing it from hallucinating or misinterpreting purely visual cues. It effectively tells the model where and how the UI structure changed, allowing the visual analysis to focus on what content was affected.

(3) Model Synergy and Dependency: The synergy between these two signals is evident in the top-performing Gemini model. However, the results also reveal a dependency on the base model's capabilities. For the less advanced Llama model, adding the textual difference signal on top of the visual one slightly degraded performance. This suggests that without a powerful reasoning engine, the structural data can act as noise rather than a helpful constraint. This finding shows that advanced VLMs like Gemini are more effective at fusing multi-modal inputs to produce a coherent and accurate judgment.

Answer for RQ3: High-cost occlusion issues can be automatically identified with high effectiveness. Our method, HCO-Eye, demonstrates this by achieving a high precision of **97.14%** and an F1-score of **93.12%** through a combined analysis of visual and structural UI changes.

VIII. Discussion

A. Implications for Developers

Although our experiments are conducted within the OpenHarmony ecosystem, our work provides developers with both a conceptual framework and a practical approach to mitigate UI overlap issues.

- **A Pitfall Guide:** The taxonomy from RQ2 serves as a pitfall guide, highlighting common patterns where defects are likely to occur. This allows developers to apply extra scrutiny during code reviews of non-hierarchical or complex layouts. Meanwhile, given that the `Ancestor-Descendant` type accounts for the majority of overlaps, developers should prioritize flattening the view hierarchy and reducing the layering of UI objects.
- **Proactive Quality Assurance:** The success of our tool, HCO-Eye, proves that subtle, semantic performance issues like high-cost occlusion are now automatable. Such tools can be integrated into CI/CD pipelines, acting as performance sentinels and shifting development from reactive bug fixing to proactive quality assurance.

B. Implications for Researchers

This study contributes a new theoretical framework and an empirical foundation to UI engineering.

- **A Robust Framework for Future Studies:** Our three-tiered taxonomy offers an objective and systematic method for classifying UI overlap based on observable properties, enabling more replicable and comparable future studies.
- **A Standard for Semantic Performance Issues:** The HCOBench benchmark is the first of its kind for this class of problem and can serve as a standard for evaluating future

tools. The success of HCO-Eye validates using VLMs for this task and highlights avenues for improvement, such as exploring advanced model architectures.

- **New Research Directions:** Our work opens up promising future avenues, including automatic defect repair, studying the impact on secondary metrics like power consumption [26], and conducting cross-platform comparative studies.

C. Implications for Platform Vendors

Our findings have direct implications for platform vendors such as the developers of OpenHarmony.

- **Improving Native UI Toolkits:** The high incidence of specific defect patterns suggests that the default behaviors of some native UI components could be made more robust. Vendors can use our findings to improve the resilience of their UI toolkits.
- **Building Intelligent IDEs:** Our work showcases the potential to build intelligent analysis tools directly into IDEs. For example, a lightweight version of HCO-Eye's logic could be integrated into DevEco Studio to provide real-time warnings for high-cost occlusion as developers write layout code, flagging these problems at the source.

D. Threats to Validity

1) Internal Validity: The random exploration strategy of our UI crawler, HapTest, may not cover issues hidden in deep business logic. This could lead to an underestimation of overlap prevalence. Additionally, the hierarchy tree obtained via uicomponent dump may lose some visual information, such as font size and precise color, which could affect the accuracy of the analysis.

2) External Validity: Our study currently focuses on the OpenHarmony ecosystem. Although the basic principles of overlap are universal, the UI frameworks and component characteristics of different platforms like Android and iOS vary, which may lead to new types of UI overlap not covered in this study. Therefore, the generalizability of our findings to other platforms needs further validation.

3) Construct Validity: Our definition of high-cost components (e.g., Video, Animation) is based on empirical induction and may not be exhaustive. A more systematic method for quantifying the rendering cost of components is needed in the future to define this problem more precisely.

IX. Related work

This research is situated at the active intersection of automated mobile GUI analysis, closely related to the fields of UI visual defect detection, GUI automation testing. This section reviews the core work in these areas to clearly position the novelty and contributions of our study.

UI Visual Defect Detection. The automated detection of UI visual defects is a critical area of software quality assurance. A approach uses computer vision to analyze single screenshots. Pioneering work like Owl Eyes [3] treated this as an image classification problem to flag defective screens. Its successor, Nighthawk [27], advances this by using object detection to draw precise boxes around defects. Other tools like Woodpecker [28] extended this idea to also include automated repair. Meanwhile, highly specialized tools like PopSweeper [29] focus on handling dynamic issues like pop-ups.

Another approach is differential testing. This method finds issues by comparing different versions of a UI. For example, dVermin [30] detects scaling problems by comparing a screen before and after system font sizes are changed. Similarly, DIFFDROID [31] finds inconsistencies by comparing an app's appearance across various devices. Despite their different strategies, these methods share a common limitation. They are all designed to identify a predefined and relatively narrow set of known issues, such as simple component occlusion or text overlap. Consequently, they struggle to detect a broader spectrum of complex layout problems. This gap highlights the need for a more general detection method.

GUI Automation Testing and Guidance. Automated testing for GUIs is an extensively studied topic. The field has evolved from early fuzz testing, which utilizes random strategies (e.g., Monkey [32], [33], [34], [35]) and model-based exploration (e.g., Stoat [36], [37], [38]), to more recent intelligence-driven approaches. This wave of innovation includes methods powered by machine learning and Large Language Models, such as AutoDroid [39], [40]. Building on this, the remarkable success of general-purpose Vision-Language Models in understanding natural images has spurred the development of powerful GUI Agents designed to automate user tasks directly from visual input, with prominent examples including MobileVLM [41], MobileAgent [42], and AppAgent [43]. However, these methods centered on exploring application behavior to enhance test coverage, complete tasks, and efficiently find functional bugs like crashes.

X. Conclusion

In this study, we present the first large-scale, systematic empirical study of the UI overlap phenomenon in HarmonyOS applications. We quantify its prevalence through automated analysis and constructed an objective, comprehensive three-tiered taxonomy, revealing its common patterns and potential risks. To address the challenging high-cost occlusion performance issue, which is difficult for traditional methods to handle, we innovatively design and validate a detection tool, HCO-Eye, based on multimodal large models. This research contributes new quantitative data, a theoretical framework for classification, and a novel VLM-driven paradigm. We believe these findings and tools will provide strong support for OpenHarmony developers in building higher-quality, higher-performance applications and will open new directions for research in the field of automated UI analysis.

Acknowledgements

This work was partially supported by the National Key Research and Development Program of China (No.2024YFB4506300) and the National Natural Science Foundation of China (No.62572024, No.62502021).

REFERENCES

[1] O. Community, "Openharmony," https://www.openharmony.cn/, 2025, accessed: 2025-01-15.

[2] L. Li, X. Gao, H. Sun, C. Hu, X. Sun, H. Wang, H. Cai, T. Su, X. Luo, T. F. Bissyandé *et al.*, "Software engineering for openharmony: A research roadmap," *arXiv preprint arXiv:2311.01311*, 2023.

[3] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Owl eyes: Spotting ui display issues via visual understanding," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 398–409.

[4] Z. Liu, "Discovering ui display issues with visual understanding," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1373–1375.

[5] H. Chen, D. Chen, Y. Yang, L. Xu, L. Gao, M. Zhou, C. Hu, and L. Li, "Arkanalyzer: The static analysis framework for openharmony," *arXiv preprint arXiv:2501.05798*, 2025.

[6] Z. Lin, M. Zhou, W. Ma, C. Chen, Y. Yang, J. Wang, C. Hu, and L. Li, "Haprepair: Learn to repair openharmony apps," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE Companion '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 319–330. [Online]. Available: https://doi.org/10.1145/3696630.3728556

[7] F. Liu, M. Zhou, Y. Zhang, T. Su, B. Sun, J. Klein, X. Gao, and L. Li, "Haptest: The dynamic analysis framework for openharmony," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE Companion '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 422–431. [Online]. Available: https://doi.org/10.1145/3696630.3728565

[8] Huawei Developer, "Layout Development Overview," https://developer.huawei.com/consumer/cn/doc/harmonyos-guides/arkts-layout-development-overview, 2025, accessed on 2025-07-28.

[9] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[10] S. Agarwal, G. Sastry, S. Seale, S. Gotesman, G. Krueger, S. Dathathri, T. McLeavey, K. Anywhere, A. Berre, C. Voss *et al.*, "GPT-4V(ision) system card," https://openai.com/index/gpt-4v-system-card/, OpenAI, Tech. Rep., September 2023, accessed on 2024-10-26.

[11] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[12] G. Comanici, E. Bieber, M. Schaekermann, I. Pasupat, N. Sachdeva, I. Dhillon, M. Blistein, O. Ram, D. Zhang, E. Rosen *et al.*, "Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities," *arXiv preprint arXiv:2507.06261*, 2025.

[13] S. Bai, K. Chen, X. Liu, J. Wang, W. Ge, S. Song, K. Dang, P. Wang, S. Wang, J. Tang *et al.*, "Qwen2. 5-vl technical report," *arXiv preprint arXiv:2502.13923*, 2025.

[14] W. Hong, W. Wang, Q. Lv, J. Xu, W. Yu, J. Ji, Y. Wang, Z. Wang, Y. Dong, M. Ding *et al.*, "Cogagent: A visual language model for gui agents," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 14 281–14 290.

[15] Y. Qin, Y. Ye, J. Fang, H. Wang, S. Liang, S. Tian, J. Zhang, J. Li, Y. Li, S. Huang *et al.*, "Ui-tars: Pioneering automated gui interaction with native agents," *arXiv preprint arXiv:2501.12326*, 2025.

[16] K. Cheng, Q. Sun, Y. Chu, F. Xu, L. YanTao, J. Zhang, and Z. Wu, "Seeclick: Harnessing gui grounding for advanced visual gui agents," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 9313–9332.

[17] C. Rawles, A. Li, D. Rodriguez, O. Riva, and T. Lillicrap, "Androidinthewild: A large-scale dataset for android device control," *Advances in Neural Information Processing Systems*, vol. 36, pp. 59 708–59 728, 2023.

[18] L. Gao, L. Zhang, S. Wang, S. Wang, Y. Li, and M. Xu, "Mobileviews: A large-scale mobile gui dataset," *arXiv preprint arXiv:2409.14337*, 2024.

[19] J. Zhang, J. Wu, T. Yihua, M. Liao, N. Xu, X. Xiao, Z. Wei, and D. Tang, "Android in the zoo: Chain-of-action-thought for gui agents," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024, pp. 12 016–12 031.

[20] K. Li, Z. Meng, H. Lin, Z. Luo, Y. Tian, J. Ma, Z. Huang, and T.-S. Chua, "Screenspot-pro: Gui grounding for professional high-resolution computer use," *arXiv preprint arXiv:2504.07981*, 2025.

[21] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the international conference on mobile software engineering and systems*, 2016, pp. 59–69.

[22] H. Developer, "Profiling - deveco device tool user guide," https://device.harmonyos.com/en/docs/documentation/guide/profiling-0000001098055350, accessed: 2025-07-29.

[23] A. Developers, "Android gpu inspector (agi)," https://developer.android.com/agi, accessed: 2025-07-29.

[24] J. Yu, H. Han, H. Zhu, Y. Chen, J. Yang, Y. Zhu, G. Xue, and M. Li, "Sensing human-screen interaction for energy-efficient frame rate adaptation on smartphones," *IEEE Transactions on Mobile Computing*, vol. 14, no. 8, pp. 1698–1711, 2015.

[25] G. Lee, S. Lee, G. Kim, Y. Choi, R. Ha, and H. Cha, "Improving energy efficiency of android devices by preventing redundant frame generation," *IEEE Transactions on Mobile Computing*, vol. 18, no. 4, pp. 871–884, 2019.

[26] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi, and H. Ye, "Application-specific performance-aware energy optimization on android mobile devices," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 169–180.

[27] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Nighthawk: Fully automated localizing ui display issues via visual understanding," *IEEE Transactions on Software Engineering*, vol. 49, no. 01, pp. 403–418, 2023.

[28] Z. Liu, "Woodpecker: identifying and fixing android ui display issues," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 334–336.

[29] L. Guo, W. Liu, Y. W. Heng, Y. Wang *et al.*, "Popsweeper: Automatically detecting and resolving app-blocking pop-ups to assist automated mobile gui testing," *arXiv preprint arXiv:2412.02933*, 2024.

[30] 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 *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

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

[32] A. Developers, "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey, 2024, accessed on: 2025-01-01.

[33] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 258–261.

[34] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical gui testing of android applications via model abstraction and refinement," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 269–280.

[35] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 481–492.

[36] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 94–105.

[37] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury, "Android testing via synthetic symbolic execution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 419–429.

[38] 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*, 2017, pp. 245–256.

[39] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J.-J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu, "Autodroid: Llm-powered task automation in android," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 543–557.

[40] 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 IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[41] Q. Wu, W. Xu, W. Liu, T. Tan, L. Liujianfeng, A. Li, J. Luan, B. Wang, and S. Shang, "Mobilevlm: A vision-language model for better intra-and inter-ui understanding," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024, pp. 10 231–10 251.

[42] 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," *arXiv preprint arXiv:2401.16158*, 2024.

[43] 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*, 2025, pp. 1–20.