GUIPilot: A Consistency-based Mobile GUI Testing Approach for Detecting Application-specific Bugs

RUOFAN LIU[†], Shanghai Jiao Tong University; National University of Singapore, China
XIWEN TEOH[†], National University of Singapore, Singapore
YUN LIN[⋆], Shanghai Jiao Tong University, China
GUANJIE CHEN, Shanghai Jiao Tong University, China
RUOFEI REN, Shanghai Jiao Tong University, China
DENYS POSHYVANYK, College of William and Mary, USA
JIN SONG DONG, National University of Singapore, Singapore

GUI testing is crucial for ensuring the reliability of mobile applications. State-of-the-art GUI testing approaches are successful in exploring more application scenarios and discovering *general* bugs such as application crashes. However, industrial GUI testing also needs to investigate *application-specific* bugs such as deviations in screen layout, widget position, or GUI transition from the GUI design mock-ups created by the application designers. These mock-ups specify the expected screens, widgets, and their respective behaviors. Validating the consistency between the GUI design and the implementation is labor-intensive and time-consuming, yet, this validation step plays an important role in industrial GUI testing.

In this work, we propose *GUIPilot*, an approach for detecting inconsistencies between the mobile design and their implementations. The mobile design usually consists of design mock-ups that specify (1) the expected screen appearances (e.g., widget layouts, colors, and shapes) and (2) the expected screen behaviors, regarding how one screen can transition into another (e.g., labeled widgets with textual description). Given a design mock-up and the implementation of its application, *GUIPilot* reports both their screen inconsistencies as well as process inconsistencies. On the one hand, *GUIPilot* detects the screen inconsistencies by abstracting every screen into a widget container where each widget is represented by its position, width, height, and type. By defining the partial order of widgets and the costs of replacing, inserting, and deleting widgets in a screen, we convert the screen-matching problem into an optimizable widget alignment problem. On the other hand, we translate the specified GUI transition into stepwise actions on the mobile screen (e.g., click, long-press, input text on some widgets). To this end, we propose a *visual prompt* for the vision-language model to infer widget-specific actions on the screen. By this means, we can validate the presence or absence of expected transitions in the implementation. Our extensive experiments on 80 mobile applications and 160 design mock-ups show that (1) *GUIPilot* can achieve 99.8% precision and 98.6% recall in detecting screen inconsistencies, outperforming the state-of-the-art approach, such as GVT, by 66.2% and 56.6% respectively, and (2) *GUIPilot* reports zero errors in detecting process inconsistencies. Furthermore, our industrial case study on applying *GUIPilot* on a trading

[†]Both authors contributed equally to this research.
[⋆]Corresponding author.

Authors' Contact Information: Ruofan Liu[†], liu.ruofan16@u.nus.edu, Shanghai Jiao Tong University; National University of Singapore, China; Xiwen Teoh[†], xiwen.teoh@u.nus.edu, National University of Singapore, Singapore; Yun Lin[⋆], lin_yun@sjtu.edu.cn, Shanghai Jiao Tong University, China; Guanjie Chen, guanjie.chen@sjtu.edu.cn, Shanghai Jiao Tong University, China; Ruofei Ren, renruofei0120@sjtu.edu.cn, Shanghai Jiao Tong University, China; Denys Poshyvanyk, denys@cs.wm.edu, College of William and Mary, USA; Jin Song Dong, dcsdjs@nus.edu.sg, National University of Singapore, Singapore.

ACM XXXX-XXXX/2025/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

mobile application shows that *GUIPilot* has detected nine application bugs, and all the bugs were confirmed by the original application experts.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; **Software prototyping**; • **Human-centered computing** → *Graphical user interfaces*.

ACM Reference Format:
Ruofan Liu†, Xiwen Teoh†, Yun Lin★, Guanjie Chen, Ruofei Ren, Denys Poshyvanyk, and Jin Song Dong. 2025. *GUIPilot*: A Consistency-based Mobile GUI Testing Approach for Detecting Application-specific Bugs . 1, 1 (April 2025), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Recent decades have witnessed the global mobile application market grow to $228.98 billion in 2023, and the market is projected to expand at an annual growth rate of 14.3% from 2024 to 2030 [61]. Rigorous GUI testing is crucial to ensure the reliability of critical mobile applications in trading, banking, and government services [35]. A typical industrial application development life-cycle includes the following stages [37]:

- **Design Stage (design mock-ups creation):** 87% of application designers utilize prototyping tools to streamline their design workflow [72]. They generate high-fidelity *mock-ups* using popular prototyping tools such as Sketch [11], Axure [8], and Balsamiq [9]. These mock-ups (1) visualize interface layouts and designs, detailing the appearance of widgets, buttons, icons, typography, and (2) illustrate how one screen can transition into another by textual description, serving as the *application specification*. Figure 1 shows an example of partial design mock-ups. This design mock-up demonstrates the expected login process, showing how the screens are interconnected through various interactive widgets.
- **Development Stage:** Based on the specifications indicated by the mock-ups, developers implement the GUI and the underlying functionalities of the widgets.
- **Testing Stage (design mock-ups validation):** Finally, application testers validate the consistency between the design mock-ups and the implementation, by writing GUI test cases and scripts. The discovered inconsistencies are reported as bugs to be fixed.

Testing mobile applications against the design mock-ups is non-trivial. First, a screen can consist of dozens of widgets and a manual comparison of two screens (one from the application and the other from the design mock-ups) can be fairly error-prone. Second, although the design mock-ups provide textual descriptions, the screen transition requires sufficient domain knowledge to complete all the triggering actions. For example, in the fourth screen of Figure 1, one must enter a password and agree to the terms by clicking the appropriate checkbox before the "Install" button becomes functional. Failing to complete these steps cannot initiate the operability of the "Install" button. Third, the mobile application can be updated frequently, the updates in the application may require a user to go through screen comparison and transition validation over and over again.

Existing GUI testing solutions are generally designed to explore more unseen scenarios and discover general bugs such as application crashes [34, 44, 45, 53, 66, 67, 76, 88, 90]. While those approaches are effective in detecting general bugs, there is little work on testing the application against the design mock-ups. The most relevant work is a GVT technique by Moran et al. [52], which detects the inconsistency between a screen in an application and a mocking screen. However, their work still suffers from the following challenges:

- **Lack of Transition Automation**: The GVT approach does not consider testing the screen transition, which is a crucial part of the specification in the design mock-ups.
- **Accurate Screen Matching**: Technically, GVT matches the widgets by *relative* position in the screen, which can miss important layout semantics. Figure 2 illustrates an example where

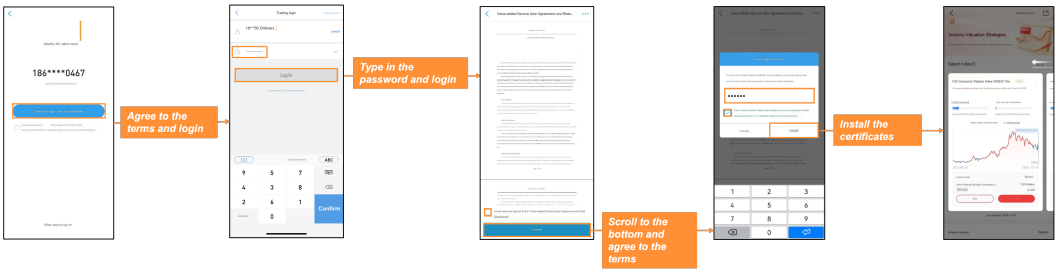


Fig. 1. A design mock-up on the login process of a trading mobile application, consisting of five screens and four transitions.

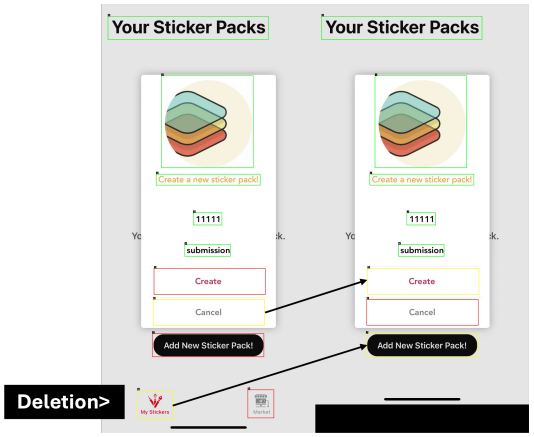


Fig. 2. A failure case of GVT. Red boxes are widgets that could not be matched. **Lines** highlight the matches.

mismatches occur when the last row of widgets is deleted. As a result, the 'Cancel' button is mismatched to the 'Create' button, and the 'My stickers' icon is mismatched to the 'Add new sticker pack' button because the matched widgets share similar relative positions on the screen. This leads to both false positives and false negatives.

In this work, we propose *GUIPilot*, an approach for detecting inconsistencies between the mobile design and its implementation, reporting both screen and process (i.e., transition) inconsistencies.

To detect screen inconsistency, *GUIPilot* abstracts every screen into a widget container where each widget is represented by its position, width, height, and type. Then, we convert the screen matching problem to an optimizable widget alignment problem by defining the partial order of widgets and the costs of replacing, inserting, and deleting widgets in a screen. By this means, we can mitigate the local matching issues in GVT and compare two screens regarding their global layout semantics.

To detect process inconsistency, we translate the GUI transition specified in the mock-up design into stepwise actions on the mobile screen (e.g., click, long-press, input text on some widgets) based on the state-of-the-art vision-language model (VLM). To mitigate the potential hallucination of VLM, we propose the *visual prompt* technique for VLM, forcing the model to infer actions only from the relevant widgets with limited action options. By this means, we can navigate to the next screen according to the design mock-ups and validate whether an expected transition can happen or be missed in the implementation.

We evaluate *GUIPilot* on 80 mobile applications with mock-up designs covering four application types and 160 design mock-ups, showing that (1) *GUIPilot* can achieve the precision of 94.5% and the recall of 99.6% in detecting screen inconsistencies, outperforming the state-of-the-art approaches, such as GVT, by 66.2% and 56.6% respectively, (2) *GUIPilot* reports zero errors in detecting process inconsistencies, and (3) *GUIPilot* is efficient in that the screen matching algorithm takes on average 0.001s and the transition takes an average of 0.19s. Further, we conduct a case study on applying *GUIPilot* on a trading application with 32 million users[1] with our industrial collaborator. The results demonstrate that *GUIPilot* detects nine application inconsistency bugs, and all the bugs have been confirmed by the application experts.

In summary, the contributions of this work are as follows:

- We propose *GUIPilot*, a solution to systematically detect inconsistencies between the design mock-ups and the mobile application implementation. To the best of our knowledge, *GUIPilot* is the first end-to-end GUI testing solution tailored for design mock-ups that are widely adopted in the industry.
- We technically address the screen matching problem regarding global layout semantics for screen consistency and the design-to-action problem for process consistency. The experiments show that we address both problems with high accuracy.
- We deliver the *GUIPilot* as a web application[2], enabling the research community and industry to conduct further research and applications.
- We conducted experiments on diverse types of mobile applications, showing the effectiveness of *GUIPilot*. Further, we show that *GUIPilot* is able to detect real design violations on an industrial trading application.

Given the space limitations, more tool demos, videos, and experimental details are available at [5].

2 Problem Statement

In this section, we provide the formal definitions and problem statements.

Screen. We consider a screen $\mathbf{s}$ as a single GUI screen, containing multiple widgets including images, buttons, text, etc. Each widget $\mathbf{w}$ is represented in $\langle x, y, w, h, t\rangle$ format. Specifically, x, y are the top-left corner coordinates of the widget on the screen, using the screenshot's top-left corner as the origin. w, h are the width and height of the widget, relative to the full screenshot size. And t is the type of the widget. We define 7 major types of widgets as detailed in Table 1. A screen contains a collection of widgets, i.e. $\mathcal{W} = \{\mathbf{w}_i | \mathbf{w}_i = \langle x, y, w, h, t\rangle\}$.

As in Table 1, we classify widgets into interactable and non-interactable. Interactable widgets include text buttons, icon buttons, combined buttons, and input boxes. A text button conveys its function through text, whereas an icon button uses an icon. A combined button integrates both text and an icon to provide a more comprehensive representation. An input box allows users to enter text. Non-interactable widgets include text views, image views, and charts.

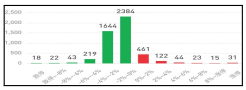
Action. On a single screen $\mathbf{s}$, one can perform a set of actions, each interacting with different widgets. We define the action space to include 7 actions: click, long press, send keys, scroll, swipe, drag and drop, and go back. An action chain $\mathbf{ac}$ is defined as a sequence of actions $\mathbf{ac} = (\mathbf{a}_1(\mathbf{w}_1), \mathbf{a}_2(\mathbf{w}_2), ..., \mathbf{a}_i(\mathbf{w}_i))$. Taking Figure 1 as an example, the action chain from screen 4 to screen 5 is $\mathbf{ac}$ = (click(the password input box), send_keys(our password), click(the agreement checkbox), click(the install button)).

Process. A process $\mathbf{p}$ is a connected directed graph $\mathbf{p} : G = (\mathcal{S}, \mathcal{AC})$, where $\mathcal{S}$ are a set of screens, and $\mathcal{AC}$ are a set of action chains. Each action chain in $\mathbf{ac} \in \mathcal{AC}$ can either lead to a transition from one screen to another or result in staying on the same screen, specifying the edges in the graph. For

[1]For the sake of anonymity, we do not reveal the name of the mobile application in the submission

[2]We have released an anonymous code repository at https://anonymous.4open.science/r/guipilot-C65C.

Table 1. Widget categories

Interactable	**Non-Interactable**
TextButton 	TextView 
IconButton 	ImageView 
CombinedButton 	Chart 
InputBox 	

example, Figure 1 consists of 5 vertices (5 screens), and they are connected by 4 edges (4 action chains or transitions).

Screen Inconsistency. We consider three types of inconsistencies as screen inconsistencies, i.e., extra widgets, missing widgets, and semantic change. Formally, we denote the set of widgets on the mock-up for a specific screen as $\mathcal{W}^{tar}$, and the set of widgets in the implementation for the corresponding screen as $\mathcal{W}$. If there exists a ground-truth matching function $f(.)$ that takes two sets of widgets and returns the sets of matched pairs, i.e. $f(\mathcal{W}^{tar}, \mathcal{W}) = \{(\mathbf{w}_i^{tar}, \mathbf{w}_j) | \mathbf{w}_i^{tar} \in \mathcal{W}^{tar}, \mathbf{w}_j \in \mathcal{W} \textit{ and they match}\}$. The inconsistency is reported when any of the conditions are met:

(i) Missing widget(s) in the implementation.

$$\exists \mathbf{w}_i^{tar} \in \mathcal{W}^{tar} \textit{ such that } \mathbf{w}_i^{tar} \notin f(\mathcal{W}^{tar}, \mathcal{W}) \tag{1}$$

(ii) Extra widget(s) in the implementation.

$$\exists \mathbf{w}_j \in \mathcal{W} \textit{ such that } \mathbf{w}_j \notin f(\mathcal{W}^{tar}, \mathcal{W}) \tag{2}$$

(iii) Semantic change in matched widgets: Two widgets are successfully paired but exhibit different semantics. For instance, their text, color, or widget type may have been altered. Let ϵ_s be the threshold for acceptable semantic changes, and let $g(.)$ represent a semantic extraction function that maps a widget to a d-dimensional representation vector. The semantic difference between two paired widgets can then be quantified as the distance between their semantic vectors:

$$\begin{aligned} &\exists (\mathbf{w}_i^{tar}, \mathbf{w}_j) \in f(\mathcal{W}^{tar}, \mathcal{W}) \textit{ such that} \\ &t_i \neq t_j \textit{ or } \|g(\mathbf{w}_i^{tar}) - g(\mathbf{w}_j)\|_2 > \epsilon_s \end{aligned} \tag{3}$$

Process Inconsistency. Process inconsistency detection is theoretically a graph comparison problem considering topological isomorphism. Let the expected process on the mock-up be denoted as $\mathbf{p}^{tar} : G = (\mathcal{S}^{tar}, \mathcal{AC}^{tar})$. $\mathcal{AC}^{tar}$ may contain ambiguous instructions, necessitating the auto-completion of the missing instructions. This results in the corrected action chain, $\mathcal{AC}^{tar}_{complete}$. Executing $\mathcal{AC}^{tar}_{complete}$ on the implementation generates a process graph on the app $\mathbf{p} : G = (\mathcal{S}, \mathcal{AC}^{tar}_{complete})$. By comparing all edges between the mock-up graph $\mathbf{p}^{tar}$ and the implementation graph $\mathbf{p}$, we identify and output the inconsistent edges:

$$\exists t, (\mathbf{s}_t^{tar}, \mathcal{AC}_t^{tar}, \mathbf{s}_{t+1}^{tar}) \in \mathbf{p}^{tar}, (\mathbf{s}_t, \mathcal{AC}_t, \mathbf{s}_{t+1}) \in \mathbf{p} \textit{ such that } sim(\mathbf{s}_{t+1}, \mathbf{s}_{t+1}^{tar}) < \epsilon_{screen} \tag{4}$$

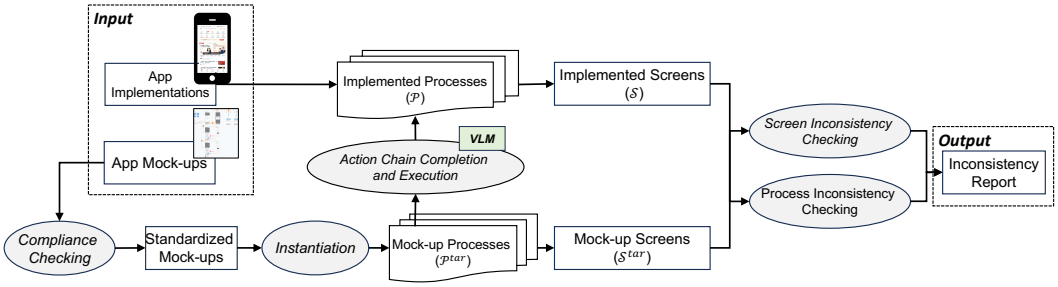


Fig. 3. Overview of the *GUIPilot* framework. It consists of three main modules: Mock-up compliance checking (Section 3.1), Screen inconsistency checking (Section 3.2), and Process inconsistency checking (Section 3.3).

3 Approach

Overview. Figure 3 presents the workflow of *GUIPilot*. The inputs consist of the implemented app running on a mobile device and a set of mock-up documents. Each mock-up describes a scenario, such as "buying exchange-traded funds (ETFs)".

First, we perform a compliance check on these raw mock-ups to parse them, ensuring each process is reformulated into a standard meta-model format (Section 3.1). Based on the action chains specified in each process, we then attempt to execute the actions on the implementation (Section 3.3). During this step, we utilize the VLM to auto-complete any implicit actions, ensuring the successful execution of the action chains. With both the mock-up processes and the executed processes on the implementation, we can perform process inconsistency checks (Section 3.3). Additionally, we compare all individual screen inconsistencies (Section 3.2). The final output is a report listing all the design violations in the implementation.

3.1 Mock-up Meta-model and Compliance Checking

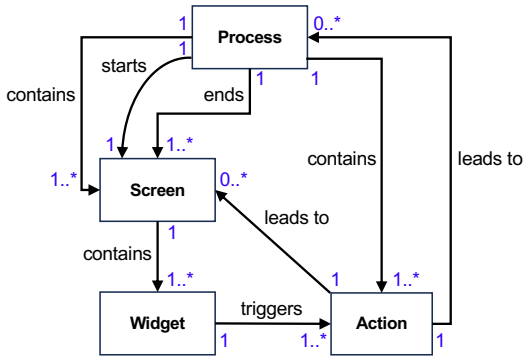


Fig. 4. Meta-model for a design mock-up.

As shown in Figure 4, our meta-model comprises four key elements: process, screen, widget, and action. Each process features a unique starting screen and at least one ending screen. A process can be represented as a connected directed graph consisting of multiple screens and the actions connecting them. The graph can also be cyclic, where the starting screen is the same as the ending screen, such

as when a user navigates to another screen and then returns to the initial one. Actions are executed on the widget elements within each screen. In this work, we consider the following action space: click, long press, send keys, scroll, swipe, drag and drop, and go back. When an action is performed, several outcomes are possible: (i) it may result in no screen transition, (ii) it may lead to another screen within the same process, or (iii) it may lead to another process. Note that, if a design mock-up does not conform to our meta-model, *GUIPilot*, as a tool, raises a compilation error for the designers to refine their mock-ups.

3.2 Screen Inconsistency Detection

In this work, we first learn a computer vision model to recognize our predefined widget types on the screens in both the design mock-ups and the application implementation. For the detected widgets, we formulate an optimizable widget alignment problem and solve it with a dynamic programming solution.

3.2.1 Widget Detection. Inspired by the previous work [17, 21, 39, 78], we adopt a vision-based approach to recognize widgets on both the mock-up screenshot and the implementation screenshot. By training a state-of-the-art object detection model [59], we can accurately identify widgets and their types as defined in Section 2. This method unifies the widget extraction process for both the mock-up and the implementation, making it more generalizable to a wide range of applications. For widget types that contain text (i.e., TextButton, CombinedButton, and TextView), we utilize an Optical Character Recognition (OCR) model to additionally extract the text content for later consistency checking. As a result, a screen is converted to a set of widgets specified with their location, shape, and type as specified in Section 2.

3.2.2 Widget Alignment. We design a widget-matching algorithm as a dynamic programming problem shown in Algorithm 1. The inputs to the algorithm are two sets of widgets from the mock-up and the implementation and the output is the set of matched pairs between these two sets. The algorithm is comprised of three critical steps:

- **Step 1: Widget *Partial Order* (Lines 2-3 in Algorithm 1).** We define *partial order* for widgets appearing on the screen. Specifically, widgets are primarily sorted by their y-coordinate in ascending order (from top to bottom). For widgets appearing in the same row, they are secondarily sorted by their x-coordinate in ascending order (from left to right). This partial ordering is derived based on our empirical observation that most widgets are positioned in a horizontal way[3].
- **Step 2: Widget Similarity Computation (Line 4-7 in Algorithm 1).** We compute the pairwise similarities between the two sets of widgets. Our similarity metric includes four components: (i) position similarity is calculated using the L1-norm distance between the (x, y, w, h) coordinates of the widgets, as originally adopted in [52], (ii) area similarity measures the differences in sizes $(w \times h)$, (iii) shape similarity assesses the differences in aspect ratios $(\frac{w}{h})$, and (iv) type similarity assigns a score of 1 if the widgets share the same class. If the widgets are of different classes (e.g., input box vs. text button), the score is down-weighted by a factor of δ. All four metrics are multiplied together to obtain the final similarity score.
- **Step 3: LCS-based Matching (Line 8-20 in Algorithm 1).** Observing that the widgets on a screen can be treated as a sequence, we convert the widget matching problem as a Longest Common Subsequence (LCS) problem [28]. Our goal is to find the *global* optimal sub-sequences of widgets from each set that achieve the highest cumulative similarity when matched. This is accomplished via dynamic programming [73], resulting in two matched widget sets, $\mathcal{W}_1^m$ and $\mathcal{W}_2^m$. By framing

[3] *GUIPilot* adopt this partial order by default, but practitioners can customize their own partial order based on the needs.

Algorithm 1: Widget Matching Algorithm $f(.)$

Input : Mock-up screen widget set $\mathcal{W}_1 = \{\mathbf{w}_i | \mathbf{w}_i = (x_i, y_i, w_i, h_i, t_i)\}$, and the implementation screen widget set $\mathcal{W}_2 = \{\mathbf{w}_j | \mathbf{w}_j = (x_j, y_j, w_j, h_j, t_j)\}$.

Output : Matched pairs between two widget sets $\mathcal{W}_1^m, \mathcal{W}_2^m$

$\mathcal{W}_1^m \leftarrow \emptyset, \mathcal{W}_2^m \leftarrow \emptyset$

// Step 1: Sort widgets by partial ordering

Sort $\mathcal{W}_1$ such that $(y_i, x_i) \leq (y_{i'}, x_{i'}), \forall i < i'$

Sort $\mathcal{W}_2$ such that $(y_j, x_j) \leq (y_{j'}, x_{j'}), \forall j < j'$

// Step 2: Compute the similarity matrix

Initialize similarity matrix $A \leftarrow \mathbf{0}_{|\mathcal{W}_1| \times |\mathcal{W}_2|}$;

for $\mathbf{w}_i \in \mathcal{W}_1$, $\mathbf{w}_j \in \mathcal{W}_2$ **do**

$$sim_{pos} = min(\frac{1}{\alpha(|x_i - x_j| + |y_i - y_j|) + |w_i - w_j| + |h_i - h_j|}, 1),\ \alpha \textit{ controls the sensitivity to position difference}$$

$$sim_{area} = \frac{\min\left(w_i h_i, w_j h_j\right)}{\max\left(w_i h_i, w_j h_j\right)}$$

$$sim_{shape} = \frac{\min\left(w_i / h_i, w_j / h_j\right)}{\max\left(w_i / h_i, w_j / h_j\right)}$$

$$sim_{type} = \mathbf{1}(t_i = t_j) + \delta \mathbf{1}(t_i \neq t_j),\ where\ 0 < \delta < 1$$

$A_{i,j} \leftarrow sim_{pos} * sim_{area} * sim_{shape} * sim_{type}$;

// Step 3: LCS-based matching

Initialize the matching matrix $M \leftarrow \mathbf{0}_{(|\mathcal{W}_1|+1) \times (|\mathcal{W}_2|+1)}$;

for $i = 2, ...|\mathcal{W}_1|,\ j = 2, ...|\mathcal{W}_2|$ **do**

$$M_{ij} \leftarrow \max\{M_{i,j-1}, M_{i-1,j}, M_{i-1,j-1} + A_{i-1,j-1}\};$$

// Backtrace the matched pairs

$i \leftarrow |\mathcal{W}_1|, j \leftarrow |\mathcal{W}_2|$;

while $i > 1$ ***and*** $j > 1$ **do**

  if $M_{ij} = M_{i-1,j-1} + A_{i-1,j-1}$ **then**

    $\mathcal{W}_1^m \leftarrow \mathcal{W}_1^m \cup \mathbf{w}_{i-1}$;

    $\mathcal{W}_2^m \leftarrow \mathcal{W}_2^m \cup \mathbf{w}_{j-1}$;

  else if $M_{ij} = M_{i-1,j}$ **then**

    $i \leftarrow i - 1$;

  else

    $j \leftarrow j - 1$;

return $\mathcal{W}_1^m, \mathcal{W}_2^m$;

the widget matching as an optimization problem, we can identify the best correspondence even in the presence of extra or missing widgets.

3.2.3 Inconsistency Report Generation. Based on the matching results, we can identify inconsistencies in screen implementation as defined in Section 2. We consider three types of violations: extra widget, missing widget, and semantic change. The first two types, i.e., extra and missing widgets, are

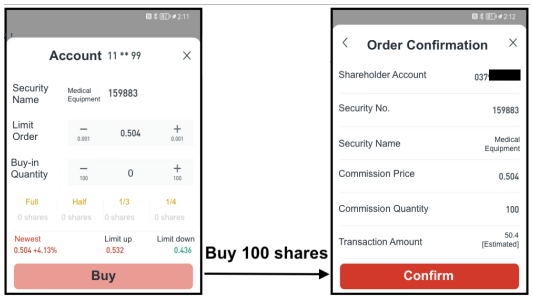


Fig. 5. Security buying process, taken from a design mock-up of an industrial trading application. The transition is described in an abstract way as "Buy 100 shares".

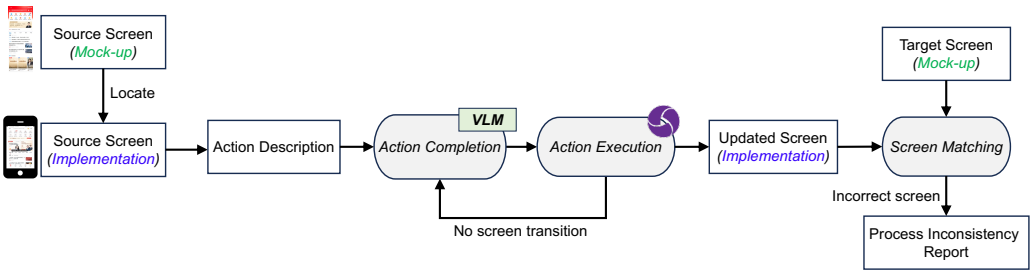


Fig. 6. Process Execution Workflow.

identified by comparing the matched sets $\mathcal{W}_1^m$ and $\mathcal{W}_2^m$ with the initial sets $\mathcal{W}_1$ and $\mathcal{W}_2$. The last type, i.e., semantic change, is computed on the matched widget pairs. If the matched pair consists of different widget types, it is considered as a change. For pairs sharing the same widget type, we follow the practice of GVT [52] to decide whether a widget is changed:

(1) For text-based widgets (TextView, TextButton, InputBox, and CombinedButton), their text values sequence similarity ratio needs to be above a threshold ϵ_{ed}.
(2) For icon-based widgets (IconButton, ImageView, InputBox, and CombinedButton), the binary color space difference must be below a threshold of ϵ_{binary}, and the top-k most frequently occurring colors' RGB difference must also be within a threshold ϵ_{color}.

Last, we relax the comparison for the widget type "Chart", i.e., we consider two charts unchanged as long as they are aligned. It is because the chart in the design mock-ups can be just an example, its content can always change in the implementation.

3.3 Process Inconsistency Detection

For each transition (in terms of a source screen $\mathbf{s}_{src}$, a target screen $\mathbf{s}_{tar}$, and their transition description $desc$) in the design mock-up (as shown in Figure 5) and the source screen in the application $\mathbf{s}^*_{src}$, we (1) translate $\mathbf{s}_{src}$ and $desc$ to a sequence of actions on $\mathbf{s}^*_{src}$ and (2) compare the resulted screen in the application to see whether $\mathbf{s}^*_{tar}$ is equivalent to the expected screen $\mathbf{s}_{tar}$.

Observing the ambiguous textual description and multi-modal design (i.e., text and image) as shown in Figure 5, we adopt the vision-language model to derive the concrete action sequence as shown in Figure 6, which can be executed by the *uiautomator* tool [1]. If a screen transition occurs, we verify whether the updated screen matches the target screen in the mock-up. A process inconsistency

Table 2. Visual prompt for process inconsistency checking with action completion. The components in blue are mutable and vary according to the current screen content.

System Prompt (Not overwritable)	
Task Objective	Given the current GUI screen, you need to return a sequence of actions to transit to the next screen.
I/O Description	The input is the **current GUI screenshot** with annotations for the widget bounding boxes (actionable widgets). The output should be a **list of actions**. Actions can be one of the following: (1) **click**(widget_id): This includes activating an input box, toggling a switch, checking a checkbox, etc. (2) **long_press**(widget_id) (3) **send_keys**(value): Once a widget is selected, one can set the value of it. (4) **scroll**(widget_id, direction, distance): Scroll to the left, right, up, or bottom by some pixels of distance. If widget_id is not specified, the default operation is to scroll the entire screen. (5) **swipe**(widget_id, direction) (6) **drag_and_drop**(widget_id1, widget_id2): Drag the widget_1 to the center of widget_2. (7) **go_back**(): This would return to the previous screen.
Few-shot Example	For example, if we have the current GUI screen with two widgets: widget_1 is a button with text "confirm", widget_2 is an input box with placeholder text as "please input the password". After I perform **click(widget_1)** action on the current GUI screen, the screen does not change, this indicates that widget_2 is unfilled. Therefore, the revised action chain should be **click(widget_2)**, **send_keys("my_password")**, **click(widget_1)**.
User Prompt (Modifiable)	
	Given the current screen and description, please provide the next immediate correct action(s). **Action Input** {Action description in Natural language. E.g. "Buy 100 shares and proceed"} **GUI Screenshot Input** {Current GUI screenshot as shown in the left.} **(Optional) Feedback** Your previous answer is incorrect, are you sure you are referring to the correct widget, or does the action exist in the action space?

is reported when they do not match. If the updated screen is correct and the process has follow-up actions, the updated screen becomes the new starting screen.

3.3.1 Visual Prompt Design. We use VLM to translate "ambiguous design" to "standard action". For example, given an ambiguous transition description like "trigger the submit button", VLM can look into the GUI screen with widget IDs, and retrieve the widget ID corresponding to the submit button (e.g., widget 1). Lastly, it returns an executable action in the format "click(widget_1)". This process incurs two major challenges:

- *Noisy information on the screen*: A screen can contain a myriad of elements, including both interactable and non-interactable widgets, as well as app-irrelevant icons. Irrelevant icons can be distracting in the action completion task and should therefore be ignored.
- *Parsability and flexibility of VLM response*: VLMs typically generate responses in a very flexible manner, which is hard to parse into programmable actions on the mobile application.

To this end, we propose a visual prompt for VLM to focus on the crucial widgets and restrict its response into a parsable format. Table 2 details our visual prompt, consisting of a system prompt shared across all samples and a user prompt that is customized based on the current GUI screen. On the one hand, we use the widget detection model (see Section 3.2.1) to highlight the interactable widgets only, attached with their widget indices. By this means, VLM can have more explicit areas to

focus on and generate responses with reference. On the other hand, we define the action space of LLM, which covers the major types of widget interactions: click, long press, send keys, scroll, swipe, and drag. Note that we mainly consider app-related interactions, not system-level interactions such as toggling the Wifi, Bluetooth, etc. To further control the randomness, we take an in-context learning approach. Specifically, we provide a few-shot example that showcases the expected response format. To save the token usage, the few-shot example is described in plain text.

In the user prompt, the query consists of both text and image modalities. The action description in natural language is fed into the LLM. And the query screenshot is annotated with interactable widgets and their IDs, this can instruct the VLM to pay special attention to the highlighted widgets only. The VLM query process can be iterative until a screen transition happens or an interaction limit is reached. If the screen has no change, we will provide feedback to VLM and ask it to reflect on its previous answer.

3.3.2 Screen Matching. Once the screen has been navigated, we need to verify whether the transited screen matches the target screen in the mock-up. Our design of the screen matching metric is based on the widget matching results in Algorithm 1. Given the updated screen and target screen, we compute the sum of similarities for all matched pairs, normalized by the total number of widgets in the target screen (Equation 5). By applying a threshold to $sim(\mathbf{s}, \mathbf{s}^{tar})$, we can determine whether the updated screen is sufficiently close to the target screen.

$$sim(\mathbf{s}, \mathbf{s}^{tar}) = \frac{\sum_{\mathbf{w}_i \in \mathcal{W}, \mathbf{w}_j \in \mathcal{W}^{tar}} A_{i,j} \cdot \mathbf{1}(\mathbf{w}_i \in \mathcal{W}, \mathbf{w}_j \in \mathcal{W}^{tar} \textit{ and they matched})}{|\mathcal{W}^{tar}|} \tag{5}$$

3.3.3 Inconsistency Report Generation. As defined in Section 2, the expected process in the mock-up is denoted as $\mathbf{p}^{tar} = (\mathcal{S}^{tar}, \mathcal{AC}^{tar})$, whereas the executed process in the implementation is $\mathbf{p} = (\mathcal{S}, \mathcal{AC}^{tar}_{complete})$. We report a process inconsistency if any of the $\mathcal{S}$ does not match $\mathcal{S}^{tar}$, i.e. $\exists \mathbf{s}_t \in \mathcal{S}, \mathbf{s}_t^{tar} \in \mathcal{S}^{tar}, sim(\mathbf{s}_t, \mathbf{s}_t^{tar}) < \epsilon_{screen}$. Here, t denotes the timestep, with ϵ_{screen} being the similarity threshold.

4 Experiments

We evaluate *GUIPilot* with the following research questions in mind:

- RQ1 (Screen Consistency Experiment): How effectively does *GUIPilot* detect *screen inconsistencies* in the public mobile application dataset?
- RQ2 (Process Consistency Experiment): How effectively does *GUIPilot* detect *process inconsistencies* in the public mobile application dataset?
- RQ3 (Component-wise Evaluation): How accurate are the critical components in *GUIPilot*? Specifically,
  - RQ3-1 What is the performance of the *widget detection* model?
  - RQ3-2 Can VLM successfully convert transition descriptions into executable actions?

Datasets. We collect 80 of the top free public applications from Google Play, from the categories of Business, Communication, Finance, and Social. We recruit four experts, each possessing at least two years of software development experience. For each app, the experts were asked to manually label two application scenarios as the mock-up processes, such as "following a social account" and "setting notification preferences". They record all the screens and intermediate action chains associated with these processes. These annotations serve as our simulated mock-ups. We derive two design mock-ups from each application, and each design mock-up comprises on average eight screens. Interested readers can refer to [7] for more examples of collected processes.

Configuration. The hyperparameters α and δ in Algorithm 1 are selected through grid search. We use the screen consistency experimental performance (as introduced in Section 4.1) as the metric to guide the selection process. The best α is chosen to 10. This scaling factor ensures that the range of sim_{pos} is more evenly distributed across (0,1]. Without this scaling, sim_{pos} tends to cluster around 1 when the widget distance is small, reducing its discriminative power. Additionally, δ is set to 0.5, meaning that when two widgets belong to different classes, their final similarity is reduced by half.

The following hyperparameters are directly adopted from GVT [52]. Specifically, we consider a string similarity ratio below $\epsilon_{ed} = 0.95$ as a text violation. When calculating color differences, we extract the Top 3 most frequently occurring colors from each widget and compute their RGB differences. A color difference exceeding $\epsilon_{color} = 0.05$ is considered a color violation. Additionally, the proportion of pixels that differ in binary color space is restricted to no more than $\epsilon_{binary} = 20\%$.

Regarding process inconsistencies, we choose the optimal screen matching threshold (ϵ_{screen} in Section 3.3.3) using the 100 mutated design mock-ups introduced in the process consistency experiment (Section 4.2). The threshold is selected to be 0.73, which achieves the highest F1-score.

4.1 RQ1: Screen Consistency Experiment

4.1.1 Setup. We mutate the screen of the simulated mock-ups to inject screen inconsistencies. Specifically, we choose the following GUI mutation types, which are common in the real application, covering 92% of the mutation cases [52]. It is a reasonable assumption that the majority of the widgets are correctly implemented. Therefore, in each mutation case, we select 5% of the widgets on the screen to be modified, which averages to 1-2 widgets per screen. This setup is the same as [52].

- **Missing widgets:** For each screen, we randomly select 5% of the widgets to delete. To simulate a realistic rendering effect, we remove the entire row containing the selected widget and shift the remaining widgets upward.
- **Extra widgets:** For each screen, we randomly insert approximately 5% additional widgets. We add complete rows for these widgets and shift the existing widgets downward accordingly.
- **Semantic change – Swapped widgets:** For each screen, we randomly select 5% of the widgets and swap them with widgets of different types to introduce semantic changes.
- **Semantic change – Text change:** For each screen, we randomly select 5% of text-based widgets and alter their text content to create semantic inconsistencies.
- **Semantic change – Color change:** For each screen, we randomly select 5% of image-based widgets and alter their colors to introduce semantic discrepancies.

4.1.2 Baseline. We choose GVT [52] as our baseline as it is the state-of-the-art screen comparison solution for mobile applications. We follow the configurations in [52] in this experiment. To test the necessity of our model design, we also include a simple baseline of direct querying VLM for inconsistency checking. Specifically, we feed the two screens, each annotated with widget IDs, to the VLM and prompt it to decide whether some of the widgets have been missed, inserted, or semantically edited.

4.1.3 Metrics. Following the metrics used in [52], we use precision, recall, Jaccard index, and classification precision as the evaluation metrics in this experiment. Let TP, FP, FN, and TP_c represent true positive inconsistency, false positive inconsistency, false negative inconsistency, and true positives with the correct type, respectively. We calculate precision, recall, Jaccard index, and

Table 3. The results of screen consistency experiment

Mutation Type	Solution	Precision	Recall	Classification Precision	Jaccard Index	Median Time (s)
Extra	*GVT*	0.793	0.899	1.000	0.690	0.001
	VLM	0.088	0.137	1.000	0.056	1.230
	GUIPilot	**0.998**	**0.986**	1.000	**0.982**	0.001
Missing	*GVT*	0.912	0.938	1.000	0.840	0.001
	VLM	0.123	0.154	1.000	0.073	1.430
	GUIPilot	**0.997**	**0.984**	1.000	**0.978**	0.001
Swap	*GVT*	0.283	0.430	1.000	0.200	0.001
	VLM	0.045	0.078	0.910	0.026	1.530
	GUIPilot	**0.987**	**0.992**	1.000	**0.971**	0.001
Text Change	*GVT*	0.998	0.999	0.981	0.960	0.001
	VLM	0.119	0.248	0.992	0.086	1.870
	GUIPilot	0.996	0.999	0.981	0.960	0.001
Color Change	*GVT*	1.000	0.999	1.000	0.990	0.001
	VLM	0.075	0.177	0.964	0.052	3.110
	GUIPilot	1.000	0.999	1.000	0.990	0.001

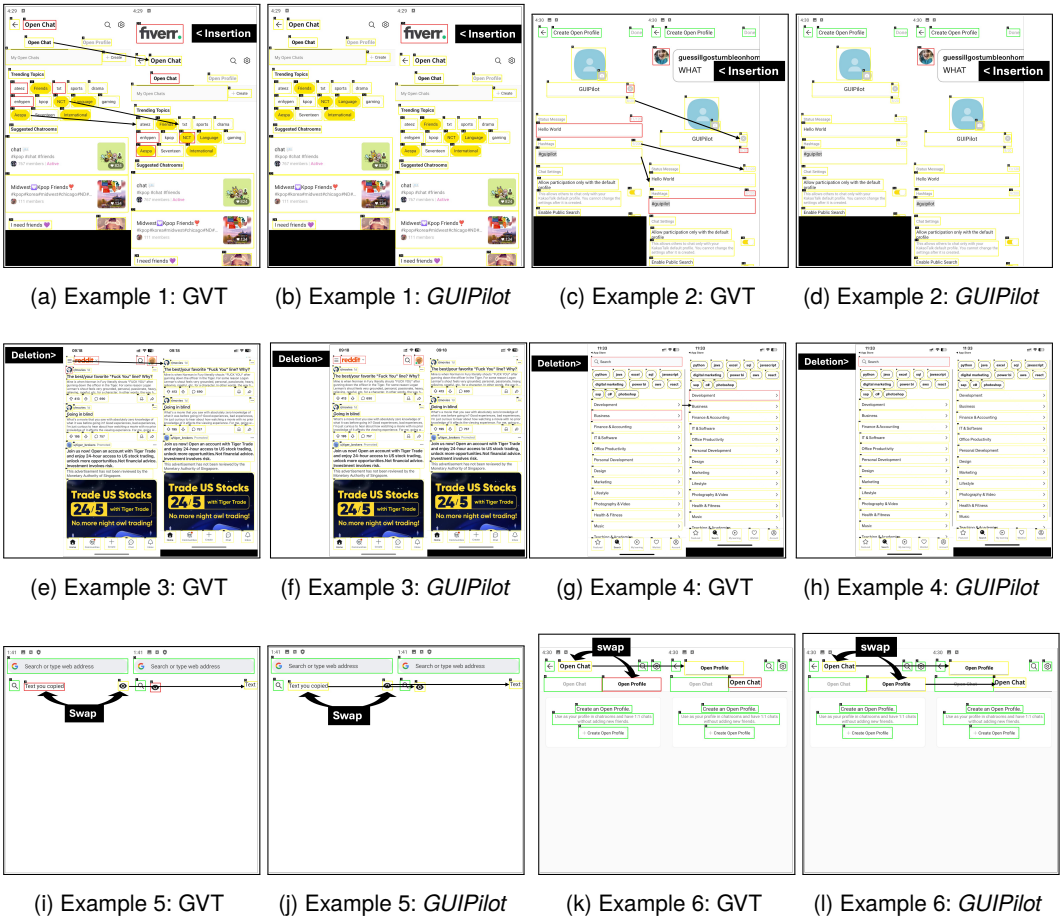
classification precision as follows:

$$pre = \frac{TP}{TP + FP}, \qquad rec = \frac{TP}{TP + FN}$$
$$J\_Index = \frac{TP}{TP + FN + FP}, \qquad cp = \frac{TP_c}{TP} \tag{6}$$

Specifically, precision pre is the number of reported true inconsistencies divided by the total number of reported inconsistencies. Recall rec is the number of reported true inconsistencies divided by the total number of inconsistencies. The Jaccard Index J_Index punishes both false negatives (unreported real inconsistencies) and false positives (false alerts). Classification precision cp is the number of real reported inconsistencies correctly identified as the correct type divided by the total number of real reported inconsistencies.

4.1.4 Results. Table 3 presents a comparative analysis between *GUIPilot* and GVT [52]. Overall, *GUIPilot* shows better performance in detecting layout violations, such as extra widgets and widget swaps. For in-place semantic changes such as color and text change, *GUIPilot*'s performance is on par with GVT. Notably, *GUIPilot* achieves these results without incurring additional runtime overhead. Theoretically, our widget alignment implements the longest-common-subsequence matching, which incurs a time complexity of $O(mn)$ where m is the total number of widgets on the mock-up and n is the total number of widgets on the implementation. Whereas GVT [52] identifies the nearest neighbor on the implementation screen for every widget on the mock-up, this has a time complexity of $O(mnlogn)$.

Additionally, directly querying VLM results in poor classification performance, with unacceptably high runtime. We observe that VLM may hallucinate non-existent inconsistencies (false positives) or identify inconsistencies but assign incorrect widget IDs (false positives and false negatives). This limitation is likely due to the resolution constraints of the visual encoder and the limitations of the local attention mechanism [40, 65]. We conduct a qualitative analysis of GUIPilot and GVT as follows, more examples can be found in our anonymous website [6].



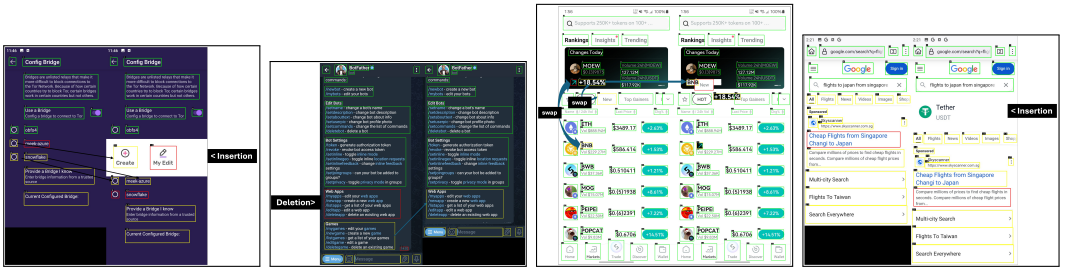
(a) Example 1: GVT (b) Example 1: *GUIPilot* (c) Example 2: GVT (d) Example 2: *GUIPilot*

(e) Example 3: GVT (f) Example 3: *GUIPilot* (g) Example 4: GVT (h) Example 4: *GUIPilot*

(i) Example 5: GVT (j) Example 5: *GUIPilot* (k) Example 6: GVT (l) Example 6: *GUIPilot*

Fig. 7. Comparison between GVT and *GUIPilot*. In each figure, the original screen is displayed on the left, and the mutated screen (after insertion, deletion, or swapping) appears on the right. Red boxes indicate extra or missing widgets. Green boxes denote widgets that are unaffected by the mutation. Yellow boxes are those who have shifted due to the mutation but can still identify a match. **Lines** highlight certain matches reported by the approach.

Why GUIPilot is better than GVT?. First, GVT is more vulnerable to location shifts. GVT predominantly depends on the relative positions of widgets on the screen for matching, making it susceptible to minor shifts in the GUI layout. In contrast, *GUIPilot* takes into account not only the widget's location but also its shape and type, enabling it to correctly match corresponding widget pairs even when minor shifts occur. As illustrated in Figure 7, examples 1-4 demonstrate that when a widget is inserted or deleted, GVT mismatches the subsequent widgets if their positions are slightly shifted. However, *GUIPilot* proves to be more robust in correctly identifying the corresponding matches. Second, GVT's strict matching threshold can overlook correct pairs. In cases where widgets are swapped (Figure 7 examples 5-6), GVT's strict matching threshold prevents it from identifying the swapping pair, as the score between them does not meet the threshold. Instead, GVT reports these as individual missing widgets or extra widgets. In contrast, *GUIPilot* can correctly match the widgets with their swapping pairs, offering a more accurate report.



(a) FP Example 1 (b) FP Example 2 (c) FP Example 3 (d) FN Example 1

Fig. 8. False positive examples (FP) and false negative examples (FN) of *GUIPilot*. In each figure, the original screen is displayed on the left, and the mutated screen (after insertion, deletion, or swapping) appears on the right. Red boxes indicate extra or missing widgets. Green boxes denote widgets that are unaffected by the mutation. Yellow boxes those who have shifted due to the mutation but can still identify a match. **Lines** highlight certain matches reported by the approach.

When GUIPilot can have false positives? We observe that *GUIPilot* can have false positives when significant layout changes happen. Although *GUIPilot* is more resilient to layout perturbations than GVT, it can still produce mismatches when substantial changes occur, and the inserted or neighboring widget happens to share a similar type and shape. As illustrated in Figure 8 (FP examples 1-2), an inserted or deleted widget may incorrectly pair with a neighboring one, causing subsequent widgets to become unpaired. These unpaired widgets are then erroneously reported as missing or extra elements. A potential remedy is to develop a more robust similarity metric that better captures the semantic and contextual similarities. For example, we can train a metric learning model to compute similarity directly from widget appearances, ensuring accurate matching even under significant layout changes.

Further, we observe that overlaying widgets can also contribute to the false positive. Widget swaps can occasionally lead to overlapping widgets (FP example 3). This overlap may cause one widget to incorrectly match with a widget from another pair or interfere with the appearance, such as altering the color of the underlying widget.

When GUIPilot can have false negatives? Similar to the reason for false positives, significant changes also contribute to false negatives. When widgets are misaligned, actual missing or extra elements may be incorrectly paired with unrelated widgets, causing them to go unreported. Further, we also observe that the failure of the widget detector can also contribute to false negatives. As shown in Figure 8 (FN example 1), the inserted widget was not detected by the object detector, resulting in a missed report.

4.2 RQ2: Process Consistency Experiment

4.2.1 Setup. We collect 160 design mock-ups across 80 applications. However, we observed that some processes exhibited frequent version updates or blocked automated interactions, thus, we retained 100 design mock-ups for the reproducibility of our experiment.

Previous studies [30, 46, 79] have identified several common root causes of incorrect screen transitions in applications, which include: (i) incorrect referencing of resource IDs, resulting in events being bound to the wrong buttons, (ii) lack of data synchronization, where global attributes—such as the user's login status—are not updated, and (iii) improper interception of events by parent or neighboring widget containers, causing touch events intended for a target widget to trigger the parent or neighboring widget instead.

Motivated by these observations, we randomly chose one screen transition represented by a <$\mathbf{s}_{src}$, $desc$, $\mathbf{s}_{tar}$> tuple for each of the 100 selected design mock-ups. We then simulate incorrect screen transitions by randomly mutating the action in a selected screen transition to produce an incorrect target screen, thereby introducing process inconsistency. Specifically, our mutation operation is chosen from one of the following:

(1) **Target Mutation (Mutate $\mathbf{s}_{tar}$).** The flow of the application returns to the previous screen instead of proceeding to the correct next screen. This simulates situations where actions have no effect due to lack of data synchronization, causing the app to unexpectedly return to the previous screen. For example, when a user adds a new event to an empty calendar page, but after the event is created, the app returns to the empty calendar page instead of displaying the updated calendar.
(2) **Source Mutation (Mutate $\mathbf{s}_{src}$).** The action is mutated to bind to a different widget. This simulates cases where the resource IDs are wrongly assigned due to similar functionality or mislabeling. For example, triggering the "Log In" event when the intended action was "Sign Up".

We then validate those 100 mutated processes in the real application following the workflow in Figure 3.3.

We evaluate whether the execution halted at the mutated screen transition due to reported inconsistencies. The outcomes were evaluated using precision and recall metrics, specifically assessing whether the screen matching correctly identified the true screen misalignments. To ensure a stable and reproducible testing environment, we employed Waydroid [12] to deploy a virtual Android device on a Linux system. All applications were installed on this device in their latest versions available during the experiments. App interactions were automated using the UiAutomator2 driver [1] and the Android Debug Bridge (ADB).

4.2.2 Results. Our approach yielded Precision and Recall scores of 100%, confirming that all introduced process inconsistencies were successfully detected. The median runtime for each screen transition was 0.193 seconds. The increase in runtime can be attributed to interaction delays inherent to in-app activities.

4.3 RQ3-1: Widget Detection Performance

4.3.1 Setup. In this experiment, we evaluate our trained widget detector's performance (Section 3.2.1). We divide the collected screens (1392 screens) from public applications into training and testing datasets at a 7:3 ratio. The training set is used to train the widget detection model, specifically, the YOLO-v8 middle [59] object detection model. The testing set, on the other hand, serves to evaluate the model's generalization performance on previously unseen screens. We use the standard metrics of mean Average Precision (mAP) and mean Average Recall (mAR) [10] in object detection tasks. These metrics are computed at the Intersection-over-Union (IoU) threshold of 0.5. When the reported bounding box overlaps with the ground-truth box by more than the IoU threshold, it is recognized as a successful match.

4.3.2 Results. Table 4 displays the class-wise performance of the widget detection model, which overall achieves a satisfactory detection rate (As a reference, the state-of-the-art performance on COCO benchmark is around 0.502 [23]). However, the accuracies for input boxes and charts are lower compared to other UI elements. The diminished accuracy for input boxes can be attributed primarily to their visual similarity to text buttons or combined buttons, which confuses the object detection model. Class confusion errors do not significantly impact our results. Even if an input box is misclassified as a text button on the mock-up screen, this misclassification would persist on the implementation screen. As a result, the two input boxes can still be aligned correctly. To remedy this, implementing a post-check that verifies whether a widget supports text input actions based on its

Table 4. Widget detection accuracy.

Super-Category	Type	**mAP**@IoU=0.5	**mAR**@IoU=0.5
Interactable	TextButton	0.585	0.715
	IconButton	0.711	0.804
	CombinedButton	0.721	0.817
	InputBox	0.446	0.511
Non-Interactable	TextView	0.677	0.758
	ImageView	0.662	0.778
	Chart	0.317	0.318
Overall		0.515	0.588

frontend code could be beneficial. Regarding charts, their detection is often compromised by the lack of clear boundaries that distinguish them from surrounding content, leading to failures in object detection. Those failure examples can be found in [4].

In this work, we use the YOLO-v8 architecture. We have also explored alternative detection methods such as other YOLO series [74, 75], RetinaNet [62], and two-stage Faster R-CNN model [60]. Figure 9 shows the performance comparison. From the comparison, we observe that the YOLO-v8 middle outperforms all other state-of-the-art models in terms of both precision and runtime efficiency while having a moderate parameter size.

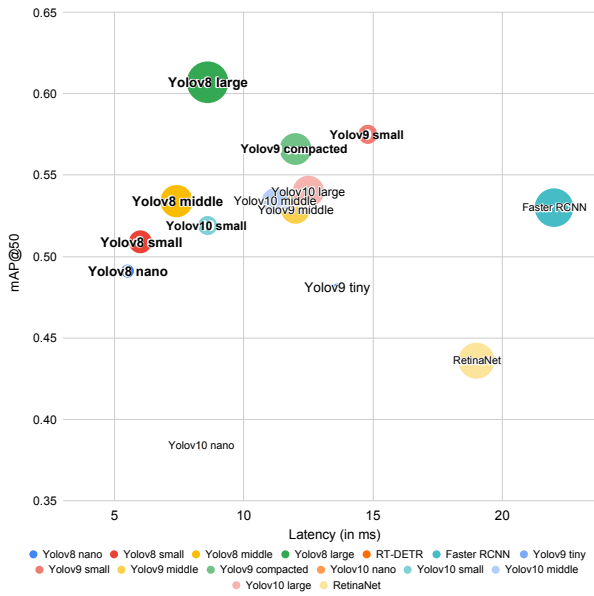


Fig. 9. Comparison of Object Detection Architectures. The Y-axis represents the mean average precision, the X-axis represents the latency per image, and the bubble size corresponds to the parameter size.

4.4 RQ3-2: VLM Action Completion Performance

4.4.1 Setup. In this evaluation, we assess the performance of the VLM agent on 100 screen transitions (shared with RQ2) to determine whether it can accurately identify relevant widgets and

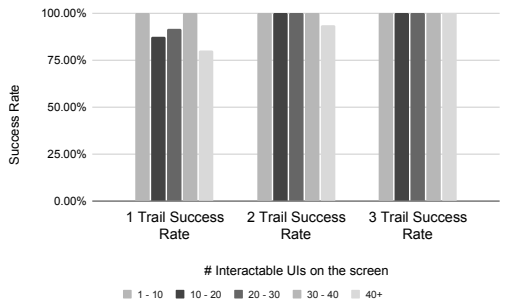


Fig. 10. VLM Action Completion Performance by UI Layout

convert actions into executable formats. For each screen transition, the inputs to the VLM include natural language commands describing actions (e.g., "click on the first suggestion", "expand menu") and a GUI screenshot where interactable widgets are indexed and highlighted.

The VLM agent is expected to: (i) accurately pinpoint the widgets on the screen that correspond to the described actions, and (ii) effectively translate these natural language actions into executable commands.

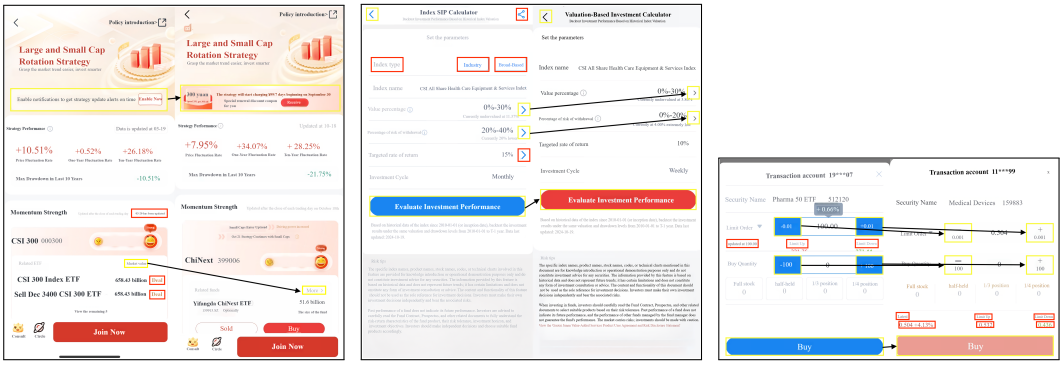
We gauge the VLM's effectiveness through the transition success rate, which measures the agent's ability to correctly execute the specified actions and achieve the anticipated screen changes. For this experiment, we employ GPT-4o, the most advanced Vision-Language Model available at the time of submission, noted for its efficiency and cost-effectiveness in handling such tasks.

4.4.2 Results. Among the 100 transitions tested, 90 achieved the correct outcome within one trial, and 99 reached the correct outcome within two trials. This suggests that incorporating self-reflection loops could further enhance the accuracy of revisions. One specific transition failed because the pertinent widget was not detected by the object detector and, therefore, was not highlighted as an interactable widget when fed into the VLM, leading to its exclusion by the VLM.

We also examine the success rate across different UI layouts. Specifically, we categorize the 100 screen transitions based on the number of widgets present on the source screen: 1-10, 10-20, 20-30, 30-40, and 40+ widgets. The results are shown in Figure 10. The darkness of the bars indicates the frequency of each UI layout, with the 10-20 widget range being the most common, followed by 20-30 widgets. The success rates are stable across different UI layouts. There is no correlation between having more UIs and a lower success rate.

5 Case Study

In collaboration with a leading investment bank and financial services provider, we obtained the real app dataset. This app is a mobile application designed to deliver comprehensive financial services to its users, including real-time stock trading, market analysis, financial news updates, portfolio management, and personalized investment advice. We conduct a qualitative study on the trading app using real mock-ups. In our collaboration with the company, we obtained 23 design mock-ups. After discarding some outdated mock-ups, we focused on 19 design mock-ups. We identified 8 instances of screen inconsistencies and 1 instance of process inconsistency. Specifically, we observed 3 additional widgets, 9 missing widgets, 3 instances of inconsistent text widgets, and 5 instances of inconsistent color schemes in the implementation. Apart from confirming the above inconsistencies in the application, the industrial experts send feedback that, the missing small widgets are particularly challenging for manual verification. This underscores the essential role of automated testing tools



(a) Instance 1: semantic change and widget missing

(b) Instance 2: semantic change and widget missing

(c) Instance 3: semantic change, widget missing, and widget extra

Fig. 11. Real screen inconsistencies reported by the *GUIPilot* on the trading App. In each figure, the mock-up is displayed on the left, and the implementation screen appears on the right. Red boxes indicate extra or missing widgets. Yellow boxes indicate semantic inconsistencies.

in identifying and addressing such inconsistencies. Those violations have been reported to and acknowledged by our industrial collaborators.

Figure 11 shows instances of different types of inconsistencies. In the first instance, the "enable notification" tab in the mock-up is replaced by a "renew subscription" tab in the implementation, the date that should be displayed in bold is not implemented, and the "market value" label is replaced with "More >" in the actual app. In the second instance, the "Index type" label is missing, the sharing icon is absent, and several color discrepancies are observed between the mock-up and the final implementation. More examples are shown on our anonymous website [3].

5.1 Discussion

Lack of Synchronization of Design Mock-ups. In the real-world App development process, the design mock-up can be deprecated with respect to the current implementation. This is because design mock-ups are not maintained with the same rigor as the mobile application. In software development, code undergoes continuous integration and continuous deployment (CI/CD), ensuring it is regularly tested, updated, and refined. Similarly, mock-ups require a systematic maintenance approach akin to CI/CD to ensure they evolve in parallel with the application to remain up-to-date. Moreover, when developers implement changes to the application, they should communicate these modifications to the design team, ensuring the mock-ups are updated accordingly. This calls for a strategic overhaul to enhance the efficacy and usability of mock-ups in development cycles.

Call for Good Practices in Mock-up Design. Our case study on industrial mock-ups has revealed significant room for improvement in the current mock-up design approach. Some important specification details are missing in the industrial design mock-ups. In the study, it is non-trivial to navigate from the home screen in the application to the starting screen specified in the design mock-ups, so that we can apply *GUIPilot*. In the study, we manually include such information in the industrial design mock-ups.

Threats to Validity: Internal Validity. Our tool relies on a third-party OpenAI service for the action completion task. Consequently, network latency and stability may affect the performance of our

proposed solution. A future direction is to distill a local large Visual-Language Model (VLM) to mitigate these issues.

Threats to Validity: External Validity. When conducting the simulation study, We use real screenshots from mobile devices to simulate the mock-ups, as we are unable to obtain the companies' actual design mock-ups. However, our *GUIPilot* can generalize to other apps as long as (i) their mock-ups pass the compliance checking of the meta-model outlined in Section 3.1 (ii) each screen transition has an accompanied description *desc* (refer to Section 3.3). It is also trivial to manually fix the mock-ups if any of the two criteria is not satisfied.

6 Related Work

GUI testing is a long-standing research topic. Some research focuses on designing automatic testing strategies to improve efficiency, code coverage, and fault localization [13, 15, 18, 26, 31, 36, 38, 47, 48, 56, 58, 68, 69, 76, 85]. Other work targets the automatic generation of test cases [29, 49, 50, 63, 64, 70, 87, 91]. Additionally, there are studies dedicated to bug replay and reporting [17, 24, 25, 27, 33, 34, 80], as well as those concentrating on code repair [14, 19, 32, 82, 86, 89].

Design Violation Detection in Mobile GUI. Seenomaly [88] uses a vision-based approach to investigate GUI animations against design guidelines, formulated as a multi-class classification task using an adversarial autoencoder. Similarly, [81] proposes 93 design guidelines and highlights prevalent violations in many Android apps. Nighthawk [45] leverages deep learning to identify UI display issues like text overlap and component occlusion due to device and software incompatibilities. The above works check GUI apps against generic design guidelines, not against specific mock-ups. The closest work to our paper is GVT [52], which provides a detailed taxonomy for static screen inconsistencies and proposes a preliminary widget matching algorithm for checking mock-up violations.

Mobile GUI Widget Detection and Widget Matching. Traditional widget detection methods rely on edge detection [51, 54] and template matching [16, 22, 57, 84] to extract salient widgets from GUI screens. Recently, widget detection has shifted to deep object detection solutions [20, 21, 78, 83], leveraging advances in computer vision for improved accuracy and reliability. For widget matching, GVT [52] identifies matched widgets based on their layout distances. Similarly, METER [55] and MAPIT [71] apply thresholds on text widgets' edit distances and graphic widgets' SIFT keypoints. However, these methods rely on pairwise criteria only, which could lead to a sub-optimal solution, making them non-robust to perturbations [49].

LLM-aided Mobile GUI Testing. Several works leverage Large Language Models (LLMs) for enhanced testing coverage, efficiency, and accuracy. QTypist [41], InputBlaster [44] and HintDroid [43] use LLMs to generate contextually appropriate text inputs. GPTDroid [42] frames mobile GUI testing as a Q&A task, using LLMs to generate and execute testing scripts. AutoDroid [77] integrates domain-specific knowledge into LLM queries for mobile task automation.

7 Conclusion

This work introduces *GUIPilot*, an innovative solution designed to detect inconsistencies between design mock-ups and actual application implementations. *GUIPilot* introduces the first comprehensive, end-to-end GUI testing solution that can detect both screen inconsistencies and process inconsistencies. We tackled the screen-matching problem by proposing an accurate widget alignment optimization, and the automatic process execution problem by introducing a visual prompt. The application of *GUIPilot* across various mobile applications has confirmed its utility and effectiveness. Specifically, in a practical scenario involving a mobile app from a financial industry, *GUIPilot* successfully identified

real design violations, underscoring its value as a tool for improving the quality and reliability of software development through enhanced mock-up fidelity.

8 Data Availability

The datasets used in this study are publicly available and can be accessed via [5]. The code used for data processing, analysis, and model training is available in the following repository [2].

References

[1] [n. d.]. Android UiAutomator2. https://github.com/appium/appium-uiautomator2-driver.
[2] [n. d.]. Anonymous Code Repository for GUIPilot. https://anonymous.4open.science/r/guipilot-C65C.
[3] [n. d.]. Anonymous Website: Case Study. https://sites.google.com/view/guipilot/case-study.
[4] [n. d.]. Anonymous Website: Failure Examples of Widget Detector. https://sites.google.com/view/guipilot/qualitative-analysis-rq3.
[5] [n. d.]. Anonymous Website for GUIPilot. https://sites.google.com/view/guipilot/home.
[6] [n. d.]. Anonymous Website: Qualitative Analysis on RQ1. https://sites.google.com/view/guipilot/qualitative-analysis-rq1.
[7] [n. d.]. Anonymous Website: Simulation Dataset Examples. https://sites.google.com/view/guipilot/dataset.
[8] [n. d.]. Axure. https://www.axure.com/.
[9] [n. d.]. Balsamiq. https://balsamiq.com/.
[10] [n. d.]. mAP (mean Average Precision) for Object Detection. https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173.
[11] [n. d.]. Sketch. https://www.sketch.com/.
[12] [n. d.]. Waydroid. https://github.com/waydroid/waydroid.
[13] Khaled Ahmed, Yingying Wang, Mieszko Lis, and Julia Rubin. 2023. ViaLin: Path-Aware Dynamic Taint Analysis for Android. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1598–1610.
[14] Ali S Alotaibi, Paul T Chiou, and William GJ Halfond. 2021. Automated repair of size-based inaccessibility issues in mobile applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 730–742.
[15] Abdulaziz Alshayban and Sam Malek. 2022. 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*. 984–995.
[16] Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, and Bo Zhou. 2015. scvRipper: video scraping tool for modeling developers' behavior using interaction data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 673–676.
[17] Carlos Bernal-Cárdenas, Nathan Cooper, Madeleine Havranek, Kevin Moran, Oscar Chaparro, Denys Poshyvanyk, and Andrian Marcus. 2022. Translating video recordings of complex mobile app ui gestures into replayable scenarios. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1782–1803.
[18] Priyanka Bose, Dipanjan Das, Saastha Vasan, Sebastiano Mariani, Ilya Grishchenko, Andrea Continella, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2023. Columbus: Android app testing through systematic callback exploration. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1381–1392.
[19] Shaoheng Cao, Minxue Pan, Yu Pei, Wenhua Yang, Tian Zhang, Linzhang Wang, and Xuandong Li. 2024. Comprehensive Semantic Repair of Obsolete GUI Test Scripts for Mobile Applications. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[20] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. 2019. Gallery dc: Design search and knowledge discovery through auto-created gui component gallery. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–22.
[21] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object detection for graphical user interface: Old fashioned or deep learning or a combination?. In *proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1202–1214.
[22] Morgan Dixon and James Fogarty. 2010. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1525–1534.
[23] Ultralytics YOLO Docs. [n. d.]. Ultralytics YOLOv8. https://docs.ultralytics.com/models/yolov8/.
[24] Sidong Feng and Chunyang Chen. 2022. Gifdroid: Automated replay of visual bug reports for android apps. In *Proceedings of the 44th International Conference on Software Engineering*. 1045–1057.
[25] Sidong Feng and Chunyang Chen. 2024. 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*. 1–13.
[26] Sidong Feng, Mulong Xie, and Chunyang Chen. 2023. Efficiency matters: Speeding up automated testing with gui rendering inference. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 906–918.
[27] Sidong Feng, Mulong Xie, Yinxing Xue, and Chunyang Chen. 2023. Read It, Don't Watch It: Captioning Bug Recordings Automatically. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2349–2361.
[28] Daniel S Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. *Commun. ACM* 18, 6 (1975), 341–343.

[29] Jiajun Hu, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2023. ωTest: WebView-Oriented Testing for Android Applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 992–1004.

[30] Yongxiang Hu, Xuan Wang, Yingchuan Wang, Yu Zhang, Shiyu Guo, Chaoyi Chen, Xin Wang, and Yangfan Zhou. 2024. AUITestAgent: Automatic Requirements Oriented GUI Function Testing. *arXiv preprint arXiv:2407.09018* (2024).

[31] Huaxun Huang, Ming Wen, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing and detecting configuration compatibility issues in android apps. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 517–528.

[32] Huaxun Huang, Chi Xu, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2023. ConfFix: Repairing Configuration Compatibility Issues in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 514–525.

[33] Yuchao Huang, Junjie Wang, Zhe Liu, Song Wang, Chunyang Chen, Mingyang Li, and Qing Wang. 2023. Context-aware bug reproduction for mobile apps. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2336–2348.

[34] Yuchao Huang, Junjie Wang, Zhe Liu, Yawen Wang, Song Wang, Chunyang Chen, Yuanzhe Hu, and Qing Wang. 2024. Crashtranslator: Automatically reproducing mobile application crashes directly from stack trace. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[35] Market IntelliX. [n. d.]. Global GUI Testing Tool Industry Research and Trends Analysis Report. https://www.marketintellix.com/sample-request/global-gui-testing-tool-industry-265845.

[36] Arun Krishna Vajjala, SM Hasan Mansur, Justin Jose, and Kevin Moran. 2024. MotorEase: Automated detection of motor impairment accessibility issues in mobile app UIs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[37] Nazar Kvartalnyi. [n. d.]. Application Development Life Cycle Explained: From Concept to Launch. https://inoxoft.com/blog/stages-of-app-development/.

[38] Yuanhong Lan, Yifei Lu, Zhong Li, Minxue Pan, Wenhua Yang, Tian Zhang, and Xuandong Li. 2024. Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[39] Yun Lin, Ruofan Liu, Dinil Mon Divakaran, Jun Yang Ng, Qing Zhou Chan, Yiwen Lu, Yuxuan Si, Fan Zhang, and Jin Song Dong. 2021. Phishpedia: A hybrid deep learning based approach to visually identify phishing webpages. In *30th USENIX Security Symposium (USENIX Security 21)*. 3793–3810.

[40] Yuliang Liu, Zhang Li, Mingxin Huang, Biao Yang, Wenwen Yu, Chunyuan Li, Xu-Cheng Yin, Cheng-Lin Liu, Lianwen Jin, and Xiang Bai. 2024. OCRBench: on the hidden mystery of OCR in large multimodal models. *Science China Information Sciences* 67, 12 (2024), 220102.

[41] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1355–1367.

[42] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. 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*. 1–13.

[43] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Yuekai Huang, Jun Hu, and Qing Wang. 2024. Unblind Text Inputs: Predicting Hint-text of Text Input in Mobile Apps via LLM. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–20.

[44] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Zhilin Tian, Yuekai Huang, Jun Hu, and Qing Wang. 2024. Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[45] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Nighthawk: Fully automated localizing ui display issues via visual understanding. *IEEE Transactions on Software Engineering* 49, 1 (2022), 403–418.

[46] Zhe Liu, Cheng Li, Chunyang Chen, Junjie Wang, Boyu Wu, Yawen Wang, Jun Hu, and Qing Wang. 2024. Vision-driven Automated Mobile GUI Testing via Multimodal Large Language Model. *arXiv preprint arXiv:2407.03037* (2024).

[47] Enze Ma, Shan Huang, Weigang He, Ting Su, Jue Wang, Huiyu Liu, Geguang Pu, and Zhendong Su. 2023. Automata-Based Trace Analysis for Aiding Diagnosing GUI Testing Tools for Android. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 592–604.

[48] Junayed Mahmud, Nadeeshan De Silva, Safwat Ali Khan, Seyed Hooman Mostafavi, SM Hasan Mansur, Oscar Chaparro, Andrian Marcus, and Kevin Moran. 2024. On Using GUI Interaction Data to Improve Text Retrieval-based Bug Localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[49] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic matching of gui events for test reuse: are we there yet?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and*

Analysis. 177–190.

[50] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *Proceedings of the 38th international conference on software engineering*. 559–570.

[51] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering* 46, 2 (2018), 196–221.

[52] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated reporting of GUI design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering*. 165–175.

[53] Kevin Moran, Mario Linares-Vasquez, Carlos Bernal-Cardenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes . In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 33–44. https://doi.org/10.1109/ICST.2016.34

[54] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 248–259.

[55] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. 2020. Gui-guided test script repair for mobile apps. *IEEE Transactions on Software Engineering* 48, 3 (2020), 910–929.

[56] Ju Qian, Yingwei Ma, Chenghao Lin, and Lin Chen. 2022. Accelerating OCR-based widget localization for test automation of GUI applications. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[57] Ju Qian, Zhengyu Shang, Shuoyan Yan, Yan Wang, and Lin Chen. 2020. Roscript: a visual script driven truly non-intrusive robotic testing system for touch screen applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 297–308.

[58] Dezhi Ran, Hao Wang, Wenyu Wang, and Tao Xie. 2023. Badge: prioritizing UI events with hierarchical multi-armed bandits for automated UI testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 894–905.

[59] Dillon Reis, Jordan Kupec, Jacqueline Hong, and Ahmad Daoudi. 2023. Real-time flying object detection with YOLOv8. *arXiv preprint arXiv:2305.09972* (2023).

[60] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2016. Faster R-CNN: Towards real-time object detection with region proposal networks. *IEEE transactions on pattern analysis and machine intelligence* 39, 6 (2016), 1137–1149.

[61] Grand View Research. [n. d.]. Mobile Application Market Size & Trends. https://www.grandviewresearch.com/industry-analysis/mobile-application-market.

[62] T-YLPG Ross and GKHP Dollár. 2017. Focal loss for dense object detection. In *proceedings of the IEEE conference on computer vision and pattern recognition*. 2980–2988.

[63] Jonathan A Saddler and Myra B Cohen. 2017. EventFlowSlicer: a tool for generating realistic goal-driven GUI tests.. In *ASE*. 955–960.

[64] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI testing for Android applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 27–37.

[65] Yiren Song, Danze Chen, and Mike Zheng Shou. 2025. LayerTracer: Cognitive-Aligned Layered SVG Synthesis via Diffusion Transformer. *arXiv preprint arXiv:2502.01105* (2025).

[66] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 245–256.

[67] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–31.

[68] Yuhui Su, Chunyang Chen, Junjie Wang, Zhe Liu, Dandan Wang, Shoubin Li, and Qing Wang. 2022. The metamorphosis: Automatic detection of scaling issues for mobile apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[69] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. 2023. Property-Based Fuzzing for Finding Data Manipulation Errors in Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1088–1100.

[70] Saghar Talebipour, Yixue Zhao, Luka Dojcilović, Chenggang Li, and Nenad Medvidović. 2021. Ui test migration across mobile platforms. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 756–767.

[71] Saghar Talebipour, Yixue Zhao, Luka Dojcilović, Chenggang Li, and Nenad Medvidović. 2021. Ui test migration across mobile platforms. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 756–767.

[72] UX Tools. 2022. Basic Prototyping. https://uxtools.co/survey/2022/basic-prototyping.

[73] Robert A. Wagner and Michael J. Fischer. 1974. The string-to-string correction problem. *Journal of the ACM (JACM)* 21, 1 (1974), 168–173.

[74] Ao Wang, Hui Chen, Lihao Liu, Kai Chen, Zijia Lin, Jungong Han, and Guiguang Ding. 2024. Yolov10: Real-time end-to-end object detection. *arXiv preprint arXiv:2405.14458* (2024).

[75] Chien-Yao Wang, I-Hau Yeh, and Hong-Yuan Mark Liao. 2024. Yolov9: Learning what you want to learn using programmable gradient information. In *European conference on computer vision*. Springer, 1–21.

[76] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. 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*. 434–446.

[77] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 543–557.

[78] Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. 2020. UIED: a hybrid tool for GUI element detection. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1655–1659.

[79] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An empirical study of functional bugs in android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1319–1331.

[80] Yanfu Yan, Nathan Cooper, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2024. Semantic GUI Scene Learning and Video Alignment for Detecting Duplicate Video-based Bug Reports. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[81] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. 2021. Don't do that! hunting down visual design smells in complex uis against design guidelines. In *2021 IEEE/ACM 43rd international conference on software engineering (ICSE)*. IEEE, 761–772.

[82] Sen Yang, Sen Chen, Lingling Fan, Sihan Xu, Zhanwei Hui, and Song Huang. 2023. Compatibility issue detection for Android apps based on path-sensitive semantic analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 257–269.

[83] Jiaming Ye, Ke Chen, Xiaofei Xie, Lei Ma, Ruochen Huang, Yingfeng Chen, Yinxing Xue, and Jianjun Zhao. 2021. An empirical study of GUI widget detection for industrial mobile games. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1427–1437.

[84] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. 2009. Sikuli: using GUI screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. 183–192.

[85] Shengcheng Yu, Chunrong Fang, Mingzhe Du, Yuchen Ling, Zhenyu Chen, and Zhendong Su. 2024. Practical Non-Intrusive GUI Exploration Testing with Visual-based Robotic Arms. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[86] Yuxin Zhang, Sen Chen, Lingling Fan, Chunyang Chen, and Xiaohong Li. 2023. Automated and Context-Aware Repair of Color-Related Accessibility Issues for Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1255–1267.

[87] Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. 2024. Learning-based Widget Matching for Migrating GUI Test Cases. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[88] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: Vision-based linting of gui animation effects against design-don't guidelines. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 1286–1297.

[89] Yanjie Zhao, Li Li, Kui Liu, and John Grundy. 2022. Towards automatically repairing compatibility issues in published Android apps. In *Proceedings of the 44th International Conference on Software Engineering*. 2142–2153.

[90] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William GJ Halfond, and Tingting Yu. 2022. Recdroid+: Automated end-to-end crash reproduction from bug reports for android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–33.

[91] Yixue Zhao, Saghar Talebipour, Kesina Baral, Hyojae Park, Leon Yee, Safwat Ali Khan, Yuriy Brun, Nenad Medvidović, and Kevin Moran. 2022. Avgust: automating usage-based test generation from videos of app executions. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 421–433.