

# Towards Efficient Record and Replay: A Case Study in WeChat

Sidong Feng  
Monash University  
Melbourne, Australia  
sidong.feng@monash.edu

Haochuan Lu  
Tencent Inc.  
Guangzhou, China  
hudsonhclu@tencent.com

Ting Xiong  
Tencent Inc.  
Guangzhou, China  
candyxiong@tencent.com

Yuetang Deng  
Tencent Inc.  
Guangzhou, China  
yuetangdeng@tencent.com

Chunyang Chen  
Monash University  
Melbourne, Australia  
chunyang.chen@monash.edu

## ABSTRACT

WeChat, a widely-used messenger app boasting over 1 billion monthly active users, requires effective app quality assurance for its complex features. Record-and-replay tools are crucial in achieving this goal. Despite the extensive development of these tools, the impact of waiting time between replay events has been largely overlooked. On one hand, a long waiting time for executing replay events on fully-rendered GUIs slows down the process. On the other hand, a short waiting time can lead to events executing on partially-rendered GUIs, negatively affecting replay effectiveness. An optimal waiting time should strike a balance between effectiveness and efficiency. We introduce WeReplay, a lightweight image-based approach that dynamically adjusts inter-event time based on the GUI rendering state. Given the real-time streaming on the GUI, WeReplay employs a deep learning model to infer the rendering state and synchronize with the replaying tool, scheduling the next event when the GUI is fully rendered. Our evaluation shows that our model achieves 92.1% precision and 93.3% recall in discerning GUI rendering states in the WeChat app. Through assessing the performance in replaying 23 common WeChat usage scenarios, WeReplay successfully replays all scenarios on the same and different devices more efficiently than the state-of-the-practice baselines.

## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

## KEYWORDS

Efficient record and replay, GUI rendering, Machine Learning

## ACM Reference Format:

Sidong Feng, Haochuan Lu, Ting Xiong, Yuetang Deng, and Chunyang Chen. 2023. Towards Efficient Record and Replay: A Case Study in WeChat. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).  
ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.  
ACM ISBN 979-8-4007-0327-0/23/12...\$15.00  
<https://doi.org/10.1145/3611643.3613880>

December 3–9, 2023, San Francisco, CA, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3611643.3613880>

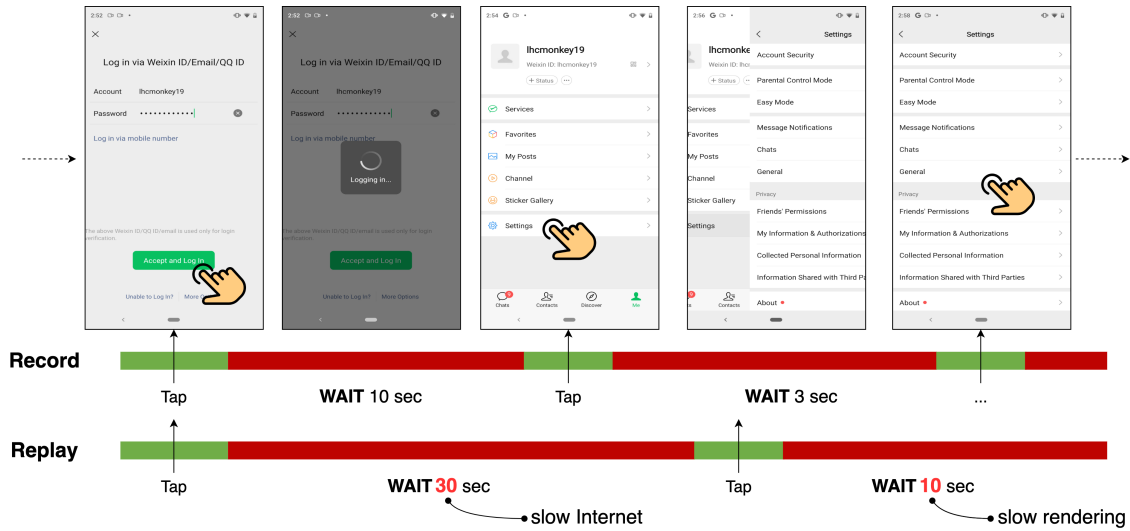
## 1 INTRODUCTION

WeChat<sup>1</sup>, with over 1.67 billion monthly active users, is among the most popular messenger apps in the world, particularly for those of Chinese origin [10]. In fact, WeChat has now evolved beyond a simple messaging app, now offering features such as banking, shopping, gaming, news browsing, and serving as a platform for third-party app development [31, 64]. As WeChat rapidly expands its features, effective app quality assurance becomes increasingly crucial and challenging. Record-and-replay tools have long been a vital approach, enabling developers to record app usage scenarios and automate their replay on various devices. Numerous record-and-replay tools have been developed by both practitioners and researchers to address this need [2, 7–9, 11, 32, 34, 38, 46, 50, 55, 72].

While these tools perform reliably in many apps, there remains a significant gap in industrial capability. One often overlooked aspect is the waiting time between replaying events. Typically, record-and-replay tools record the time delay between events and automatically replay them on devices with the same elapsed time. However, our study of nine WeChat usage scenarios reveals that a fixed time elapse may not be practical due to two main reasons. First, GUI loading for replaying may take longer due to Internet-related factors, which are common in the WeChat app such as logging in, sending messages, etc. Second, replaying on older devices may require more time due to decreased rendering efficiency and lower processing capability. These issues of executing replay events on incompletely rendered GUIs impede successful replay on the same device (with a 45% failure rate) and different devices (with a 63% failure rate).

To ensure effective replaying, WeChat developers usually manually set event time delays to a considerably longer waiting time — about 10 times longer than the recorded duration. However, this manual setting process can be time-consuming and error-prone for developers, taking an average of 10 minutes to set up 5 events, according to the WeChat developers. Moreover, replay becomes inefficient with idle waiting on fully rendered GUIs, slowing down the testing process. Due to budget constraints and market pressures in the industry, development teams often face limited testing time. Consequently, the more efficient the replay tools, the more devices

<sup>1</sup><https://www.wechat.com/>



**Figure 1: Example of record and replay events. Green bars represent event triggers, while red bars represent the inter-event waiting time. A fixed waiting time captured during the recording may not be sufficient for replaying, as dynamic factors such as slow internet or delayed rendering can affect the process.**

can be tested, increasing the likelihood of discovering bugs and ultimately improving the quality of the app release.

Towards that target, we draw inspiration from previous work [29] to accelerate automated app testing with GUI rendering inference. While the app under testing is mostly ideal, the replay tool has to wait until the GUI finishes rendering before moving to the next event. With this in mind, we introduce WeReplay, a lightweight record-and-replay tool designed to dynamically schedule replay events by distinguishing between fully rendered and partially rendered GUIs. To record the events to WeChat, we leverage the app inspector tool WEditor to retrieve the widget information and parse it to the testing script. To replay the events in the testing script dynamically, we adopt a deep learning method to model the visual information from the GUI screenshot for inferring the GUI state. Specifically, we first develop a tailored method to collect a large-scale binary GUI dataset from WeChat, comprising 4,485 fully rendered and 6,171 partially rendered GUIs. Next, we adopt a small but efficient Convolutional Neural Network (CNN) based approach to discern the GUI rendering state. To implement our approach, ensuring events are sent only when the GUI is fully rendered, we establish a synchronization framework that streams real-time GUI screenshots and schedules replay events based on GUI rendering inference. Notably, one advantage of our approach is its purely image-based nature, making it easily deployable in other industrial apps.

To evaluate the accuracy of our approach, we carry out an experiment on the GUI screenshots from the WeChat app. Our approach outperforms nine state-of-the-art baselines and two ablation baselines in predicting GUI rendering states, achieving 92.1% precision, 93.3% recall, and 92.7% F1-score, respectively. We also carry out an experiment to demonstrate that our tool can accelerate the replaying process without sacrificing effectiveness by replaying 23 usage scenarios from the WeChat app. Given a recorded usage scenario

from one device, our tool can accurately replay it on both the same device and different devices in less time, compared to the baselines and industrial solutions.

The contributions of this paper are as follows:

- We propose WeReplay, a practical record-and-replay tool designed to adaptively adjust the waiting time between replaying events, for speeding up the replaying process.
- We conduct a motivational empirical study to understand the prevalence of waiting time issues in record-and-replay for WeChat, which serves as the foundation for our research.
- We perform comprehensive experiments, including an evaluation of WeReplay’s performance and its industrial record-and-replay practice in WeChat, to demonstrate the accuracy, efficiency, and effectiveness of our tool.

## 2 MOTIVATIONAL STUDY

Many record-and-replay tools have been developed by both practitioners and researchers to assist developers in easily recording and replaying complex usage scenarios of apps. To assess the practicality of these record-and-replay tools in industrial apps, we conducted a pilot study by applying state-of-the-practice tools to the most popular social media app, WeChat.

### 2.1 Experimental Setup

We collected an experimental dataset of nine common usage scenarios from WeChat to evaluate whether the record-and-replay tools can accurately record and replay such scenarios. These scenarios were identified by WeChat developers and through our hands-on experiences using the app, with an average of 6.4 events per scenario. An example of a common usage scenario, “opening setting page in WeChat” is illustrated in Figure 1, including inputting a username and password to log in to the app and accessing settings on the personal page.

**Table 1: Overview of devices.**

Device	Resolution	Size (inch)	OS Version	CPU Processor
Xiaomi Mix2S	1080x2160	5.99	9.0	Snapdragon 845
Huawei Nova2S	1080x1920	5.50	8.0	Kirin 659
Vivo Y3	720x1544	6.35	9.0	Helio P35
Google Pixel4a	1080x2340	5.81	11.0	Snapdragon 730G

To record and replay these common usage scenarios, we chose the state-of-the-practice tool SARA [33] for two reasons. First, SARA is widget-sensitive, meaning that the recorded events are based on the attributes of the widget (e.g., resource id). This approach is more robust for replaying on multiple devices with different resolutions and GUIs compared to recording coordinates [70]. Second, SARA’s recording is timing-sensitive, as it automatically captures the time between events and utilizes this information during the replay process. We did not adopt other publicly available record-and-replay tools (e.g., Robotium [72], Culebra [2], etc.) due to their impracticality in industrial apps or the requirement for costly investments [44].

## 2.2 Record and Replay on Same Device

To understand the practice of record-and-replay in WeChat, we initially conducted a small pilot study focused on replaying usage scenarios recorded by SARA on the same device. We utilized a Xiaomi Mix2S running Android 9.0 as the testing device, which is widely used in previous studies [71]. In total, we obtained 9 usage scenarios replays captured using screencast. Two authors manually examined the states in the video replays to evaluate the correctness of each replay.

55% of the scenarios were successfully replayed. We manually examined the failure cases and identified one common cause. The waiting time between events is dynamically affected by internet bandwidth. For example, consider a scenario where an app is logged and recorded in an environment with excellent internet connectivity, resulting in a shorter waiting time (i.e., 10 seconds in Figure 1). However, when replaying the scenario in an environment with poor internet connectivity, the replay fails due to the increased waiting time required to trigger the next event (i.e., 30 seconds in Figure 1). WeChat developers further confirmed this phenomenon in recording and replaying industrial apps like WeChat, which frequently rely on internet connectivity.

## 2.3 Record and Replay on Different Devices

To evaluate the reproducibility of scenarios across different devices, we first recorded the scenarios using a Xiaomi Mix2S. We then replayed the events on three devices to ensure usability for a wide variety of users in practice. Table 1 provides details of these devices, covering different resolutions, sizes, operating system versions, and CPU processors. These devices are widely used in previous studies for record-and-replay experiments [57, 71]. In total, we obtained 27 replays and manually assessed the reproducibility of each one.

We discovered that 11%, 33%, and 66% of the scenarios were successfully replayed for the Huawei Nova2S, Vivo Y3, and Google Pixel4a, respectively. The former two devices replayed fewer scenarios than the latter, particularly the Huawei Nova2S, which could

**Table 2: A study of waiting time in replaying.**

Method	1x wait	2x wait	5x wait	10x wait
Reproducibility	11%	33%	77%	100%
Time	23.2s	49.4s	112.1s	217.5s

only replay 11% of the scenarios. The main reason is that a fixed amount of waiting time between events recorded on a more advanced device may not be suitable for replaying on older devices. This can be attributed to factors such as decreased rendering efficiency and lower processing capability often associated with older devices. WeChat developers further confirmed that unexpected device lagging can significantly impact the performance of scenario replays.

## 2.4 Industrial Solution & Motivation

WeChat developers confirm the significance of the waiting time settings for recording and replaying on both the same and different devices. They also emphasize that setting arbitrary time delays between events is undesirable, as they are likely to fail due to insufficient loading time between events. To ensure that the recorded scenario is 100% replayable on any device, WeChat developers need to manually set the time delay of events to a relatively long waiting time, usually 10x longer than the recorded waiting time, as shown in Table 2. However, this approach raises two practical issues. First, the manual setting can be time-consuming, taking more than 10 minutes on average for a 5-event scenario, according to WeChat developers. Second, the replay process is inefficient, frequently stagnating on fully loaded GUIs and slowing down the replaying process, i.e., taking over 3.5 minutes on average to replay 5 events.

While the app under testing is mostly ideal, the replaying has to wait until the GUI finishes rendering before proceeding to the next event. In this context, it is worth developing an effective and efficient method to dynamically adjust the waiting time during replaying. The underlying challenge is to infer GUI rendering states, differentiating between partially rendered and fully rendered GUIs. Inspired by the fact that humans can easily classify these GUIs visually, we propose to identify the GUI rendering states with visual cognitive techniques.

## 3 WEREPLAY TOOL

This paper presents a simple but effective approach for adaptively adjusting the waiting time based on GUI states. Drawing inspiration from the previous approach, AdaT [29], which aimed to accelerate automated app testing by categorizing GUI rendering states, we explore its practical implementation in the industrial app WeChat and further expand the approach for recording and replaying.

The overview of our tool is shown in Figure 2. In the recording phase, we record all input events, including widget attributes, actions, and input data, and parse them into the testing script. In the replaying phase, we synchronously capture the GUI screenshots and detect their current state. If the GUI is fully rendered, we schedule the next replaying events; otherwise, we explicitly wait for rendering to complete.

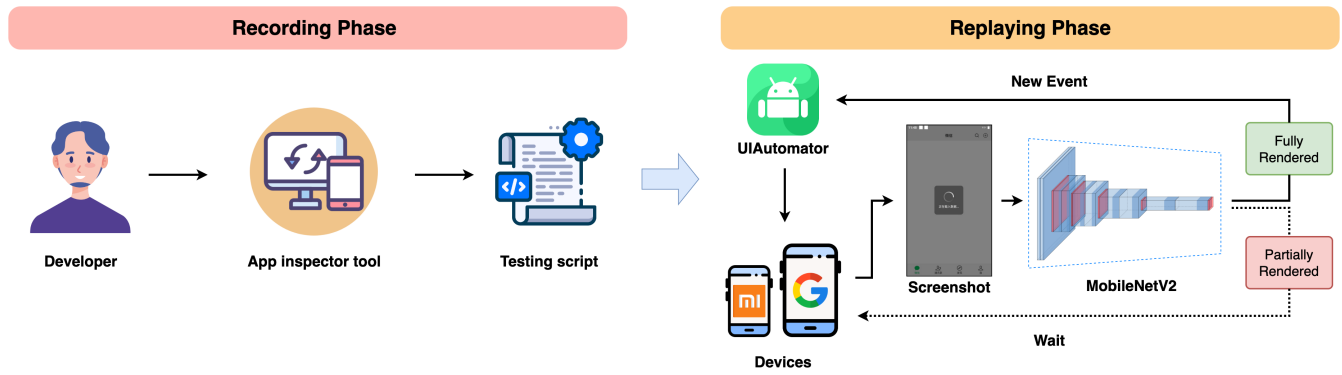


Figure 2: The workflow of our approach.

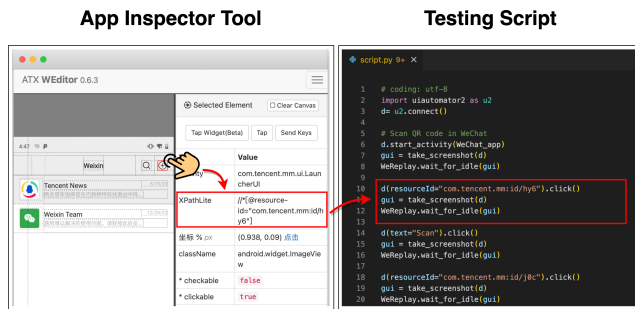


Figure 3: The testing script recorded by app inspector tool.

### 3.1 Recording Phase

There are many available tools for recording testing scenario scripts, either manually [64] or automatically [33, 72]. To ensure the effectiveness of testing scripts at the industrial level, we manually record the rich variety of events, including the widget attributes, actions, and input text. Figure 3 illustrates how a user event is transformed into a script. In detail, we first use the app inspector tool WEditor [15] to inspect the widget information from the interface, as shown in Figure 3. Given the widget information, we then parse it into executable testing scripts using the widely-used automated app testing framework UIAutomator2 [5]. It is worth noting that other tools can also be employed for this purpose.

### 3.2 Replaying Phase

The foundation of our approach is to utilize a lightweight CNN-based model to classify the WeChat GUI rendering state during replaying. To achieve this, we first introduce a novel approach for collecting a large-scale dataset consisting of partially rendered and fully rendered GUIs in the WeChat app. Using this data, we propose a CNN-based model to distinguish between WeChat GUI states.

**3.2.1 WeChat Data Preparation.** The foundation of training deep learning models is big data. To begin, we first manually annotate the record-and-replay screencasts from Section 2, while gaining an understanding of the GUI rendering states in WeChat. Based on this understanding, we introduce three tailored data augmentation

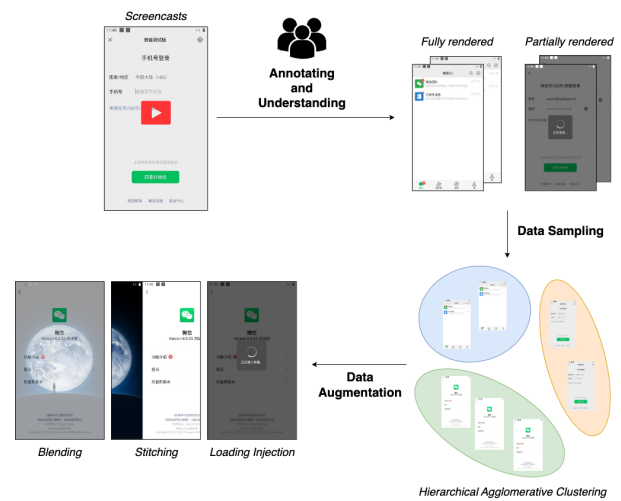


Figure 4: The pipeline of WeChat data preparation.

techniques, e.g., stitching, blending, and loading injection, to automatically synthesize more data. A pipeline for preparing WeChat data is illustrated in Figure 4.

**Annotating and understanding.** We recruited two developers as annotators through WeChat’s internal slack channel. According to the pre-study background survey, they have previously labeled one UI/UX-related dataset (e.g., GUI element bounding box). To ensure accurate annotations, the process started with initial training. First, we asked them to read the previous GUI rendering study [29] and a document [3] that outlines the GUI rendering process. Second, we provided an example set of annotated GUIs where the rendering states have been labeled by authors. This enforces a deeper understanding of the GUI rendering states. Third, we asked them to pass an assessment test, which includes a set of test GUIs. Finally, we asked them to manually check 23,331 GUIs from 36 record-and-replay screencasts. During the manual examination process, we identified three types of GUI states following the Card Sorting [62] method:

*a) Transiting State:* As depicted in Figure 5(a), one state is transiting to the next state. The transition between states takes time, causing GUIs to overlap or merge during the process. This issue



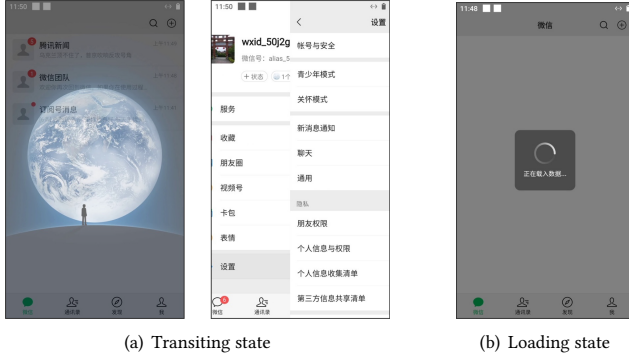


Figure 5: Examples of partially rendered state.

often occurs in devices with subpar performance and hardware acceleration defects, resulting in unexpectedly prolonged loading times. Replaying events in this state may lead to unforeseen errors.

*b) Loading State:* As illustrated in Figure 5(b), this state displays an explicit loading indicator in the GUI, such as a spinning wheel, linear progressing bar, textual hint, etc. It explicitly indicates the process or rendering is in progress and is often used for secure data transformations, such as logging into accounts, transferring money, uploading a file, etc. During the explicitly loading state, the GUI is non-interactive to replay events.

*c) Fully Rendered State:* A fully rendered state represents a GUI rendered completely with all resources loaded and displayed.

**Data sampling.** For each GUI state group, we observe that many GUIs are duplicated. This is because the rendering changes relatively slowly between consecutive GUI frames from the screencasts. To prevent this bias, we propose a paradigm that uses hierarchical agglomerative clustering (HAC) [52] to sample the GUI frames from the GUI groups.

The detail of HAC is shown in Algorithm 1. That is, each GUI frame is initially considered as a single-element cluster (Lines 1-7). At each step of the algorithm, the two most similar clusters are merged into a new cluster (Line 10). This procedure is iterated until all the clusters reach the agglomerative similarity threshold  $\epsilon$  (Lines 12-16).

The main parameter in this algorithm is the metric used to compute the similarity value of GUI frames. To achieve this, we adopt the commonly-used perceptual metric, Structural Similarity Index (SSIM) [67], which produces a per-pixel similarity value related to the local difference in average value, variance, and correlation of luminances. The SSIM score ranges between 0 and 1, with a higher value indicating a strong level of similarity. Considering the properties of sparseness and distinctness in data sampling, we empirically set the threshold  $\epsilon$  to 0.9. After automated clustering and sampling, we obtain a dataset with 4,485 fully rendered GUIs and 3,159 partially rendered GUIs.

**Data augmentation.** Training an effective deep learning model requires a large amount of input data [45]. However, gathering and labeling relevant GUIs can be both time-consuming and labor-intensive. Data augmentation techniques, such as cropping, rotation, color space transformations, etc., are commonly employed to overcome this constraint [60]. The closer the synthetic data

### Algorithm 1: Hierarchical Agglomerative Clustering

---

**Input** : GUI frames  $F = \{f_1, f_2, f_3, \dots, f_n\}$ ;  
Agglomerative similarity threshold  $\epsilon$ ;

**Output**: Sampled frames  $S$

```

/* Initialize */
1 let  $com[0..m, 0..n]$  be new computation table ;
2 initialize  $com \leftarrow 0$  ;
3 for  $i \leftarrow 1$  to  $n$  do
4   for  $j \leftarrow 1$  to  $n$  do
5      $com[i][j] \leftarrow SSIM(f_i, f_j)$ 
6   end
7 end
8  $S \leftarrow []$ ;
/* Iteratively merge the new cluster until reaching threshold
 $\epsilon$  */
9 while  $F.length > 1$  do
10   $(f_i, f_j) \leftarrow argmax(com)$  ;
11   $F.remove(\{f_i, f_j\})$  ;
12  if  $com[i][j] \geq \epsilon$  then
13     $S.append(f_i)$  ;
14  else
15     $F.append(f_i \cup f_j)$  ;
16  end
/* Update computation table to new clusters */
17 for  $i \leftarrow 1$  to  $F.length$  do
18   for  $j \leftarrow 1$  to  $F.length$  do
19      $com[i][j] \leftarrow SSIM(f_i, f_j)$ 
20   end
21 end
22 end
23 return  $S$ 

```

---

is to the real one, the better the model's performance. Therefore, we develop three novel data augmentation methods based on our aforementioned understanding of the WeChat GUIs.

*a) Stitching:* As illustrated in Figure 5(a), one particular transiting state occurs when one GUI slides to the next. To synthesize this, we employ image stitching, which combines two GUIs with segmented states to create a sliding view. Specifically, we randomly crop the GUIs horizontally, as GUI sliding typically occurs in this direction. Next, we generate a seamless connection between the two cropped GUIs, resulting in a synthesized GUI.

*b) Blending:* As illustrated in Figure 5(a), another transiting state occurs when one GUI fades out while the next GUI fades in, causing overlap between the two GUIs. To synthesize this, we employ image blending, which linearly combines two GUIs in a gradient transition. Specifically, we utilize a weighted average method, where the opacity of each GUI is adjusted according to a random transition curve, to ensure a smooth transition between the GUIs.

*c) Loading Injection:* As illustrated in Figure 5(b), the loading state typically displays an explicit loading symbol, such as a spinning wheel. To synthesize the loading state, we inject the loading symbol into the GUIs. Additionally, we apply varying intensities of shadow effects to the GUIs to create a more dynamic loading state.

**3.2.2 GUI Rendering State Classification.** To identify whether a GUI is fully rendered, allowing the replaying tool to execute the next event, we adopt an implementation of MobileNetV2 [59]. This model distills the best practices in convolutional network design into a simple architecture that offers competitive performance while maintaining low parameters and mathematical operations to reduce computational costs and memory overhead. This advanced network design accelerates image classification, which is the ultimate goal of this work to efficiently discriminate between GUI rendering states.

Specifically, we adopt a more advanced depthwise separable convolution, combining one  $3 \times 3$  convolution layer and two  $1 \times 1$  convolution layers to capture essential information from images. We first use a  $1 \times 1$  pointwise convolution layer to expand the number of channels in the input feature map. Then, we use a  $3 \times 3$  depthwise convolution layer to filter the input feature map and a  $1 \times 1$  convolution layer to reduce the number of channels of the feature map. In order to improve the performance and stability between layers, we borrow the idea of residual connection in ResNet [35] to help with the flow of gradients. After the convolution layer, we add Batch Normalization (BN) [39] to standardize the feature map. Finally, the activation function, Rectified Linear Unit (ReLU), is added to increase the nonlinear properties of the classifier function and of the overall network without affecting the features.

For detailed implementation, we fine-tune the model from previous work [29] using our tailored data to adjust the parameters in the classification layers for the specific industrial app, WeChat. We adopt the stride of 2 in the depthwise convolution layer to downsample the feature map. We use ReLU6 defined as  $y = \min(\max(0, x), 6)$ , for the first two activation layers because of its robustness in low-precision computation [36]. A linear transformation (also known as Linear Bottleneck Layer) [59] is applied to the last activation layer to prevent ReLU from destroying features. The momentum in the BN layer is set as 0.1. To make our training more stable, we adopt Adam as optimizer [42], and binary CrossEntropyLoss as the loss function [53]. Moreover, to optimize the training model, we apply an adaptive learning scheduler, with an initial rate of 0.01 and decay to half after 10 iterations. For data preprocessing, we resize the GUI screenshots to  $768 \times 448$ . We implement our model based on the PyTorch framework [54]. Note that the hyper-parameter settings are determined empirically by a small-scale experiment.

**3.2.3 Model Deployment.** To enable the model to efficiently provide feedback on the GUI rendering state to the replaying tool, synchronization between the GUI and the replaying tool is necessary. In detail, we use Android Debug Bridge (adb) [6] to capture and transmit real-time GUI screenshots. Once the screenshot is received, we decode it into a PyTorch tensor [54]. This tensor is then fed into our trained GUI state classification model to infer the current GUI's rendering state. If the GUI is fully rendered, we proceed to replay the new event; otherwise, we explicitly wait for the next screenshot. To prevent excessive time consumption due to prolonged resource loading or incorrect model predictions, we set a maximum waiting threshold. This waiting threshold is empirically set at 60 seconds based on a small pilot study.

## 4 EVALUATION

In this section, we describe the procedure we used to evaluate our approach in terms of its accuracy, effectiveness, and efficiency to accelerate the record-and-replay process. To achieve our evaluation, we formulate the following three research questions:

- **RQ1:** How accurate is our model in classifying WeChat GUI rendering state?
- **RQ2:** How effective and efficient is our tool in replaying WeChat usage scenarios on the same device?
- **RQ3:** How effective and efficient is our tool in replaying WeChat usage scenarios on different devices?

For **RQ1**, we present some general performance of our model in inferring WeChat GUI states and the comparison with state-of-the-art baselines. For **RQ2**, we carry out experiments to check if our tool can speed up the automated replaying of the usage scenarios in WeChat on the same device, without sacrificing the effectiveness. For **RQ3**, we conduct experiments to evaluate the effectiveness and efficiency of our tool to replay on different devices with diverse screen sizes and operating system versions.

### 4.1 RQ1: Performance of WeChat GUI State Classification

**4.1.1 Experimental Setup.** To answer RQ1, we first evaluated the ability of our model MobileNetV2 (in Section 3.2.2) to accurately differentiate between fully rendered GUIs and partially rendered GUIs. To accomplish the evaluation, we followed the procedure to generate the dataset outlined in Section 3.2.1. We collected 23,331 GUI screencasts and annotated 9,119 partially rendered GUIs and 14,212 fully rendered GUIs. Due to the presence of duplicates in the dataset, we employed a clustering algorithm to eliminate these redundancies, resulting in 3,159 partially rendered GUIs and 4,485 fully rendered GUIs. Next, we used tailored data augmentation methods to synthesize 3,012 partially rendered GUIs. In total, we obtained 6,171 partially rendered GUIs and 4,485 fully rendered GUIs as our experimental dataset. Regarding our training-testing data split, we split this 10k GUIs in the ratio of 8:1:1 for the training, validation, and testing sets, respectively. The resulting split has 8k GUIs in the training dataset, 1k GUIs in the validation dataset, and 1k GUIs in the testing dataset. The model was trained in an NVIDIA GeForce RTX 2080Ti GPU (16G memory) with 20 epochs for about 2 hours.

**4.1.2 Metrics.** Since we formulated our problem as an image classification task, we adopted three widely-used metrics i.e., precision, recall, and F1-score, to evaluate the accuracy of the model inference. Precision is the proportion of GUIs that are correctly predicted as fully rendered among all GUIs predicted as fully rendered.

$$\text{precision} = \frac{\#GUIs \text{ correctly predicted as fully rendered}}{\#All \text{ GUIs predicted as fully rendered}}$$

Recall is the proportion of GUIs that are correctly predicted as fully rendered among all fully rendered GUIs.

$$\text{recall} = \frac{\#GUIs \text{ correctly predicted as fully rendered}}{\#All \text{ fully rendered GUIs}}$$

**Table 3: Performance comparison with baselines**

Methods	Precision	Recall	F1-score
SIFT+SVM	0.706	0.751	0.727
SIFT+KNN	0.592	0.639	0.614
SIFT+RF	0.678	0.670	0.673
SURF+SVM	0.634	0.625	0.629
SURF+KNN	0.577	0.601	0.588
SURF+RF	0.606	0.659	0.631
ORB+SVM	0.659	0.699	0.678
ORB+KNN	0.596	0.616	0.605
ORB+RF	0.636	0.667	0.651
AdaT	0.859	0.852	0.855
WeReplay w/o aug	0.893	0.906	0.899
WeReplay	<b>0.921</b>	<b>0.933</b>	<b>0.927</b>

F1-score (F-score or F-measure) is the harmonic mean of precision and recall, which combine both of the two metrics above.

$$F1 - score = \frac{2 \times precision \times recall}{precision + recall}$$

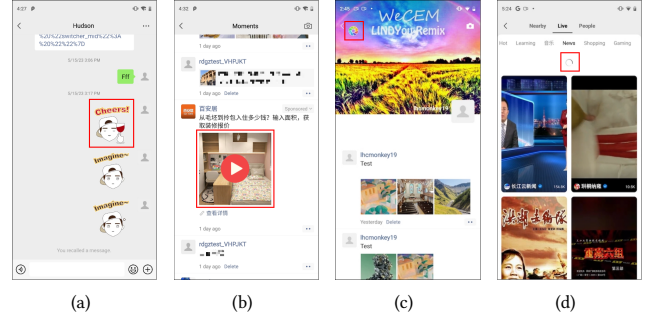
For all metrics, a higher value represents better performance.

**4.1.3 Baselines.** We set up 9 state-of-the-art baseline methods, that are widely used in image classification tasks as the baselines to compare with our model. They first extract visual features from the GUI screenshots and then employ a machine learner for the classification. In detail, we adopted three types of feature extraction methods used in machine learning, e.g., Scale-invariant feature transform (SIFT) [51], Speed up robot features (SURF) [17], and Oriented fast and rotated brief (ORB) [58]. With these features, we applied three commonly-used machine learning classifiers, e.g., Support Vector Machine (SVM) [43], K-Nearest Neighbor (KNN) [41], and Random Forests (RF) [18], for classifying the GUI rendering state. The combination of three types of image features and three classification learning algorithms generated a total of 9 baselines.

In addition, we also added 2 ablation studies as our baselines to demonstrate the advantage of our model. First, we experimented with the off-the-shelf pre-trained model AdaT [29] to see its general performance in WeChat GUIs. It uses a convolutional neural network MobileNetV2 to extract the visual features and trains on 79k GUIs from the 9.7k Google app to classify the fully rendered and partially rendered states. Second, we investigated the contribution of our data augmentation methods in Section 3.2.1, namely WeReplay w/o aug, to see the performance of our model trained without 3,012 (40%) additional data.

**4.1.4 Result.** Table 3 illustrates the performance of our approach in classifying the fully rendered GUIs and partially rendered GUIs in the industrial app WeChat. Our approach significantly outperforms other baselines, achieving a 28.7%, 30.8%, and 29.8% increase in recall, precision, and F1-score, respectively, compared to the best machine learning baseline (SIFT+SVM). We observe that deep learning-based methods perform much better than machine learning methods, primarily because machine learning lacks feature introspection, which is crucial as GUI rendering state features vary.

Compared to the pre-trained deep learning baseline AdaT, our model further improves recall, precision, and F1-score by 8.1%, 6.2%,

**Figure 6: Examples of bad cases in GUI state prediction.**

and 7.2%, respectively. This suggests that fine-tuning a pre-trained model enables it to better recognize specific features, such as the distinct GUI characteristic in the industrial app WeChat. Consequently, this enhances the model’s ability to classify GUI states more accurately. Additionally, we observe that applying tailored data augmentation methods further improves our model’s performance, increasing recall, precision, and F1-score by 2.7%, 2.8%, and 2.8%, respectively. This indicates that with more training data, the model’s GUI rendering classification capability improves. In the future, we plan to collect more diversified GUIs from the WeChat app to enhance the model’s performance.

Albeit the good performance of our model, we still make incorrect predictions for some GUI screenshots. We manually examined these inaccuracies and identified two common causes. First, within the industrial app WeChat, GUIs may contain dynamic assets such as videos and gifs, as seen in the gif message in Figure 6(a) and the advertisement in Figure 6(b). In these cases, although the GUIs are fully rendered, they may be misclassified as partially rendered GUIs due to animation loading. Second, some representative features of loading in WeChat are too small and inconspicuous to be recognized, even by human eyes, for example, the tiny circular progress bar embedded in the image in Figure 6(c) and embedded in the background in Figure 6(d).

## 4.2 RQ2: Performance of Recording and Replaying on Same Device

**4.2.1 Experimental Setup.** To answer RQ2, we evaluated the ability of our tool to effectively and efficiently record and replay usage scenarios on the same device. Throughout our three months study, we had access to WeChat developers and periodically asked them for the scenarios on their hands-on experiences using the app. In total, we gathered 23 fundamental usage scenarios in the WeChat app as our experimental dataset. On average, each scenario contained 5.3 events, covering various WeChat functionalities such as messaging, games, shopping, etc. Note that we did not use the usage scenarios collected in Section 2 for this evaluation, as they were utilized for model training and could potentially result in a data leakage problem [40].

We recorded and replayed the usage scenarios on Xiaomi Mix2S running Android version 9.0. The device was allocated with 8 dedicated CPU cores, 6GiB of RAM, and discrete graphics cards for

minimal mutual influences caused by disk I/O bottlenecks and CPU-intensive graphical rendering.

**4.2.2 Metrics.** To measure the performance of our approach, we employed reproducibility as the evaluation metric, i.e., whether the method can successfully replay the usage scenarios. The higher the reproducibility score, the better the approach can replicate the events to reproduce the scenarios. Since the ultimate goal is to speed up the replaying process, we also measured the time for replaying. For the replaying time, a lower time cost represents faster replaying of the recorded scenarios.

**4.2.3 Baselines.** We set up two methods as our baseline to compare with our approach. First, we adopted the state-of-the-art method SARA [33]. Specifically, SARA employs a heuristic technique to record the waiting time elapsed between events. Second, we followed the practical solution in WeChat, as discussed in Section 2, extending the waiting time by 10 times to create an industrial baseline, referred to as *10x wait*. Note that we did not evaluate other publicly available tools like Espresso [8] and Culebra [2], as they are not sensitive to waiting time, e.g., setting arbitrary inter-event time, which prevents effective evaluation on industrial apps.

**4.2.4 Result.** Table 4 shows detailed results of the time and reproducibility for each usage scenario. On average, our tool WeReplay takes 18.45 seconds to reproduce all the scenarios. We observe that, on one hand, SARA, with a short fixed waiting time, can only replay 39.1% of the scenarios. The failure cases are due to dynamic Internet loading, i.e., resources taking longer to load than the recorded waiting time. On the other hand, the industrial solution of extending the waiting time by 10 times significantly slows down the replaying process, taking an average of 152.99 seconds per scenario. In contrast, our tool can replay all scenarios (100%) in less time (18.45 seconds), achieving 60.9% more successful reproductions compared to SARA, while saving 88% more time than 10x wait with the same reproduction rate. As a result, WeReplay can expedite the replaying process without sacrificing reproducibility, saving a significant amount of time in long-term industrial testing involving hundreds or thousands of events.

### 4.3 RQ3: Performance of Recording and Replaying on Different Devices

**4.3.1 Experimental Setup.** To answer RQ3, we evaluated the ability of our tool to effectively and efficiently record and replay events on different devices. In detail, we used the Xiaomi Mix2S to record the 23 fundamental usage scenarios in the WeChat app outlined in Section 4.2.1. To replay these usage scenarios, we employed three different devices, including Huawei Nova2S, Vivo Y3, and Google Pixel4a. Details of the device information can be seen in Table 1, covering diverse screen resolutions, sizes, operating systems, and processors.

**4.3.2 Metrics.** To measure the performance of our tool, we employed two evaluation metrics, i.e., reproducibility (replay success) and time (replay time).

**4.3.3 Baselines.** We adopted the state-of-the-art SARA [33] and industrial solution *10x wait* in Section 2 as the baselines to compare with our tool.

**Table 4: Performance comparison of replaying on the same device. “R” denotes scenario reproducibility. “T” denotes the time to replay in seconds.**

Scenario	Events	SARA		10x wait		WeReplay	
		R	T	R	T	R	T
Open Moments	3	✓	15.62	✓	149.13	✓	12.39
Open GameCenter	4	✓	23.16	✓	229.61	✓	22.21
Open Channels	4	✓	22.01	✓	224.33	✓	20.67
Open e-Wallet	4	✗	10.20	✓	96.45	✓	18.96
Open Pay QRCode	4	✗	10.92	✓	101.17	✓	13.32
Open Collections	4	✓	10.13	✓	98.36	✓	9.36
Search Info	6	✗	16.24	✓	161.74	✓	36.07
Search Game	6	✗	56.96	✓	553.21	✓	40.12
Search New Friend	6	✗	12.81	✓	113.25	✓	20.37
Delete Friend	5	✓	21.89	✓	203.93	✓	14.20
Enter Shopping	4	✓	16.35	✓	159.01	✓	12.32
Recommend Friend	6	✗	9.51	✓	90.43	✓	12.07
Change Username	8	✗	13.92	✓	133.86	✓	15.57
Subscribe Stickers	5	✗	9.11	✓	89.70	✓	23.94
Search Stickers	8	✗	15.72	✓	155.71	✓	30.88
Change Profile Photo	6	✓	23.82	✓	219.95	✓	24.63
Video Call	5	✗	6.52	✓	64.49	✓	10.07
Audio Call	5	✗	6.21	✓	56.62	✓	9.22
Unfollow Official Account	7	✗	12.42	✓	117.37	✓	23.63
Group Chat	6	✗	10.26	✓	99.67	✓	14.19
Send Message	5	✓	15.03	✓	143.44	✓	12.21
Send Stickers	5	✓	11.79	✓	112.88	✓	11.84
Send Picture	7	✗	15.07	✓	144.46	✓	16.19
Average	5.35	39.1%	15.89	100%	152.99	100%	18.45

**4.3.4 Result.** Table 5 presents the detailed results of the replaying performance across the three devices. Our tool WeReplay, successfully replays all (100%) of the scenarios in a shorter time, with a median of 19.55 seconds. In contrast, the baseline SARA only replays 21.7%, 34.8%, and 47.8% of the scenarios for Huawei Nova2S, Vivo Y3, and Google Pixel4a, respectively. The failure scenarios are due to the delayed rendering process for the devices, indicating that more time is needed for rendering. The industrial solution can also replay all of the scenarios, but it takes much longer, with an average of 157.41 seconds, making it 10 times slower than our tool. This is because it indiscriminately extends the waiting time between events by 10 times, while most of this time is spent idly waiting. In contrast, our tool WeReplay utilizes a deep learning model to identify the GUI rendering state, dynamically scheduling events if the GUI is fully rendered, or explicitly waiting otherwise, achieving efficiency without compromising replay capability.

## 5 DISCUSSION

We had discussed the limitations of our tool in misclassifying the WeChat GUI rendering state due to dynamic assets (Section 4.1.4). In this section, we discuss the industrial implication and threats to validity of our tool.



**Table 5: Performance comparison of replaying on the different devices. “R” denotes scenario reproducibility. “T” denotes the time to replay in seconds.**

Scenario	Huawei Nova2S						Vivo Y3						Google Pixel4a					
	SARA		10x wait		WeReplay		SARA		10x wait		WeReplay		SARA		10x wait		WeReplay	
	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T
Open Moments	✓	18.77	✓	150.14	✓	18.99	✓	16.81	✓	147.63	✓	13.79	✓	16.31	✓	145.56	✓	12.73
Open GameCenter	✗	27.96	✓	241.14	✓	25.58	✗	25.72	✓	237.77	✓	24.91	✓	16.12	✓	235.52	✓	22.37
Open Channels	✓	28.64	✓	243.01	✓	24.64	✓	24.73	✓	231.12	✓	20.97	✓	24.97	✓	232.44	✓	18.65
Open e-Wallet	✗	13.94	✓	104.91	✓	16.36	✗	12.65	✓	103.49	✓	16.33	✗	11.48	✓	101.76	✓	13.95
Open Pay QRCode	✗	13.01	✓	107.99	✓	21.23	✗	11.97	✓	107.89	✓	18.08	✗	11.50	✓	104.97	✓	16.17
Open Collections	✗	12.75	✓	108.76	✓	15.52	✓	11.03	✓	104.99	✓	10.19	✓	10.93	✓	103.65	✓	8.89
Search Info	✗	20.46	✓	169.02	✓	40.96	✗	18.15	✓	168.63	✓	34.66	✗	17.31	✓	163.47	✓	20.86
Search Game	✗	67.78	✓	564.98	✓	51.03	✗	61.51	✓	559.98	✓	44.69	✗	18.79	✓	555.72	✓	29.90
Search New Friend	✗	16.77	✓	122.56	✓	20.24	✗	13.22	✓	118.75	✓	18.09	✓	14.79	✓	114.02	✓	13.98
Delete Friend	✓	23.75	✓	215.97	✓	19.52	✓	22.19	✓	210.60	✓	16.38	✓	20.01	✓	203.19	✓	15.11
Enter Shopping	✓	19.33	✓	164.98	✓	14.07	✓	17.59	✓	164.34	✓	11.77	✓	18.89	✓	162.06	✓	11.15
Recommend Friend	✗	12.65	✓	95.09	✓	16.94	✗	11.21	✓	93.68	✓	12.96	✗	10.46	✓	92.18	✓	10.78
Change Username	✗	15.89	✓	141.40	✓	17.50	✓	15.92	✓	136.04	✓	15.12	✓	14.80	✓	135.88	✓	13.51
Subscribe Stickers	✗	14.75	✓	91.57	✓	31.44	✗	10.14	✓	89.94	✓	26.01	✗	9.14	✓	89.07	✓	26.91
Search Stickers	✗	17.88	✓	163.18	✓	35.78	✗	17.04	✓	150.61	✓	32.06	✗	16.50	✓	148.82	✓	30.85
Change Profile Photo	✗	29.76	✓	228.95	✓	27.90	✗	26.55	✓	223.74	✓	24.49	✗	25.05	✓	220.99	✓	21.66
Video Call	✗	8.93	✓	71.93	✓	20.74	✗	7.22	✓	68.62	✓	10.98	✗	6.52	✓	67.02	✓	7.60
Audio Call	✗	9.11	✓	62.07	✓	16.77	✗	7.23	✓	60.49	✓	9.69	✓	6.93	✓	55.48	✓	7.72
Unfollow Official Account	✗	17.36	✓	126.75	✓	25.04	✗	14.12	✓	120.88	✓	23.04	✗	13.57	✓	112.27	✓	19.20
Group Chat	✗	12.88	✓	102.97	✓	17.86	✗	11.25	✓	100.47	✓	17.53	✗	11.67	✓	99.68	✓	15.96
Send Message	✓	19.02	✓	144.98	✓	17.72	✓	15.31	✓	144.44	✓	12.64	✓	14.54	✓	142.52	✓	12.64
Send Stickers	✗	14.41	✓	120.74	✓	16.83	✓	13.01	✓	119.78	✓	11.81	✓	12.59	✓	117.04	✓	11.32
Send Picture	✗	17.79	✓	151.35	✓	17.93	✗	15.19	✓	148.74	✓	15.89	✗	15.07	✓	142.28	✓	15.63
Average	21.7%	19.72	100%	160.62	100%	23.07	34.8%	17.38	100%	157.46	100%	19.17	47.8%	14.69	100%	154.16	100%	16.41

## 5.1 Industrial Implication

Although many state-of-the-art automated recording tools exist [16, 33, 72], we chose manual recording for usage scenarios in the industrial app WeChat due to two practical lessons learned. First, automated recording of widgets in hybrid apps like WeChat (apps built with a combination of native and web technologies) presents steering challenges, as some widgets are rendered by WebView as HTML elements, which cannot be found in the view hierarchy. Second, games in WeChat’s mini-programs [49] are often developed with game engines (e.g., Unity [14]), making it impossible to automatically record the widget. To ensure the effectiveness of recording usage scenarios in the WeChat app, we generate them manually by inspecting the widget in the inspector tool WEditor. In the future, we plan to conduct a comprehensive empirical study of the cases where the widgets cannot be automatically recorded and will attempt to develop an engineering effort to address this issue.

Another industrial implication involves supporting record and replay for different platforms, such as iOS. The results in Section 4.2.4 and Section 4.3.4 demonstrate strong performance in recording and replaying on Android devices. Although we focus on the Android platform in this study for brevity, our approach could be extended to other platforms. As the GUI rendering process in iOS exhibits minimal differences compared to Android, our approach could be adapted to it with a reasonable amount of engineering effort.

## 5.2 Threats to Validity

Threats to internal validity may arise from the manual labeling of the training and testing dataset for the GUI rendering classification model. To mitigate any potential subjectivity or errors, we provided the annotators with a training session and a qualifying test before labeling. We instructed them to independently annotate without any discussion, and we reached a consensus on the finalized dataset. Although there may still be some noise in the data, training a deep learning-based model with a sufficient amount of high-quality data can tolerate a small amount of noise [45, 63].

In our experiments evaluating our tool, threats to external validity may arise from the representativeness of industrial usage scenarios depicted in our experimental set. To mitigate this threat, we conducted experiments on 23 real-world usage scenarios provided by WeChat developers. While performing additional experiments with more scenarios would be ideal, our experimental set of scenarios represents a reasonably fundamental set of tests with different functionalities, illustrating the relative performance of our tool. Another potential confounding factor concerns the representativeness of the record-and-replay devices used in the evaluation. To mitigate this threat, we employed four devices with different resolutions, sizes, operating systems, and processors, as outlined in Section 2. These devices are practically used as testing devices in WeChat and widely studied in previous studies [57, 71]. One more potential threat concerns the generalizability of our approach to other industrial apps. In this study, we focus on the WeChat app,

using WeChat GUIs to train the GUI rendering classification model to help speed up the record and replay process in practice. Our approach relies only on GUI screenshots, which should be easily adapted to other industrial apps with customized datasets.

## 6 RELATED WORK

The main quality of our study is the utilization of the GUI rendering state to accelerate the record-and-replay process in the industrial app WeChat. Therefore, we review the related work in two main areas: 1) record and replay for industrial apps, and 2) efficiency support for testing.

### 6.1 Record and Replay for Industrial Apps

The primary goal of record-and-replay tools is to record an app's execution to facilitate automatic replay. RERAN [32] is one of the earliest record-and-replay tools that use the Linux kernel. Specifically, it captures low-level events with the ADB command `getevent` by reading logs in `/dev/input/event*` files and uses the command `sendevent` to replay events. However, the low-level events recorded are tightly coupled to the hardware, making it difficult to reconstruct high-level gestures for replay, such as zoom, pinch-in, etc. MobiPlay [56] introduces a client-server architecture, involving a client app running on a mobile device and a target app running on the server to record events. It requires a custom OS to record and replay, which may lead to incompatibilities between the OS and the device, thereby violating industry constraints. Monkeyrunner [11], a desktop tool developed by Google Inc., provides an API for writing Python programs to record and replay keystroke coordinates on Android devices. Subsequent tools like Mosaic [34], HiroMacro [9], VALERA [38], and RepetiTouch [13] capture events in a similar manner. Nevertheless, the underlying recording and replaying based on pixel coordinates are often prone to failure due to minor GUI changes across diverse devices.

Consequently, several researchers [24, 30] have developed widget-sensitive record-and-replay tools. For example, Robotium [72] is derived from the Selenium web browser automation tool and records events only if GUIs are controlled by the app's main process. Whereas, many industrial apps may use different processes to avoid compatibility issues on various platforms. One example of how WeChat uses other processes to create GUI widgets is through its mini-programs, which are embedded frameworks within the WeChat app. Ranorex [12] is a commercial test automation tool that supports recording events based on widgets through app instrumentation. However, instrumentation requires sophisticated accessibility or GUI automation APIs [22, 23] and continuous updates in sync with the app and different operating systems [47, 65], making it incompatible with industrial apps. The official Android Studio IDE introduces Espresso [8], which leverages source code analysis to record events based on widgets by attaching a debugger to the app, but it still requires developers to manually set the time delay of events, which can be troublesome and error-prone.

Guo et al. [33] introduce a practical record-and-replay tool SARA, that satisfies the industry requirements for widget-sensitive and time-sensitive recording and replaying. Specifically, it proposes a self-replay mechanism to record user event information while capturing timestamps to infer the waiting time between replay events.

However, a fixed waiting time may not accurately replay events according to our analysis (less than 55% for the same device and 37% for different devices) in Section 2. First, the waiting time can be indeterminate, depending on the internet, which is frequently used in industrial apps like WeChat. Second, for cross-device replaying, the waiting time can be significantly dependent on the performance of the devices, with lower-performing devices typically requiring more waiting time. In contrast, our tool does not record a fixed waiting time but leverages a novel deep learning-based model to infer GUI rendering state, dynamically adjusting the waiting time to schedule events on fully rendered GUIs. The empirical evaluations of record and replay in the WeChat app confirm the practicality of our tool.

### 6.2 Efficiency Support for Testing

Many works have attempted to improve infrastructure support for efficient mobile testing. Hu et al. [37] propose AppDoctor, which instruments target apps using event handler invocations to quickly identify potential sequences of error-triggering GUIs. Song et al. [61] enhance the efficiency of AppDoctor by leveraging direct invocations. Wang et al. [66] propose an Android tool, Toller, that injects into the testing device to efficiently access GUI layout and execute events. In contrast to these infrastructure support methods, our goal is to accelerate record and replay by using adaptive waiting times, i.e., scheduling testing events for efficiency improvement.

Adaptive waiting time is a common practice for efficient record and replay on the web. Selenium [4] introduces a feature called `Explicit Wait`, which instructs the testing driver to wait for a specified amount of time until the presence of widgets. Similar to Selenium, many tools incorporate this feature for mobile testing, such as Appium [1], UIAutomator [5], etc. Specifically, these tools verify the presence of widgets by fetching the view hierarchy of the GUI. However, subsequent studies [48] find that the fetched GUI views may be out of sync, leading to events on misaligned or invalid objects. While some studies [19–21, 25–28, 68, 69] conduct UI modeling based on view hierarchy, they only check the validity of the GUI view hierarchy, not resource loading, which limits the effectiveness of replaying. In contrast, we leverage the GUI as a whole with visual information to dynamically adjust the waiting time and schedule events when the GUI is fully rendered, which is analogous to human viewing and interaction. Following the line of previous work [29], we integrate the GUI rendering classification model into the industrial app WeChat, offering insight into its potential for recording and replaying usage scenarios.

## 7 CONCLUSION

Record-and-replay is essential for ensuring quality assurance in the industrial app, WeChat. Despite the numerous record-and-replay tools available, the waiting time between replaying events is often overlooked. A short waiting time may hinder the effectiveness of replaying, while a long waiting time may reduce efficiency. To address this, we propose a practical record-and-replay tool WeReplay, that employs a lightweight image-based approach to adaptively adjust the waiting time based on GUI rendering inference. Given the real-time streaming on GUI, WeReplay uses a deep learning model

to infer the rendering state and adjust event scheduling, replaying events when the GUI is fully rendered. Experiments demonstrate the performance of our approach in improving the efficiency and effectiveness of recording and replaying within the WeChat app.

In the future, we plan to continue improving WeReplay's performance by collecting more GUI data from the WeChat app to enhance the accuracy of our model. While WeReplay's motivation stems from WeChat, its adoption is not limited to this app, as it relies solely on GUI screenshots, which are easily obtainable in industrial apps. We plan to explore its potential usage in other industrial apps and systematically evaluate its performance.

## REFERENCES

- [1] 2022. Appium. <http://appium.io/>.
- [2] 2022. Culebra. <https://github.com/dmilano/AndroidViewClient/wiki/culebra>.
- [3] 2022. Rendering - Android Developers. <https://developer.android.com/topic/performance/rendering>.
- [4] 2022. Selenium. <https://www.selenium.dev/>.
- [5] 2022. UI Automator. <https://developer.android.com/training/testing/other-components/ui-automator>.
- [6] 2023. Android Debug Bridge (adb). <https://developer.android.com/studio/command-line/adb>.
- [7] 2023. appetizer-toolkit. <https://github.com/appetizerio/appetizer-toolkit>.
- [8] 2023. Espresso Test Recorder. <https://developer.android.com/studio/test/espresso-test-recorder.html>.
- [9] 2023. HiroMacro Auto-Touch Macro. <https://play.google.com/store/apps/details?id=com.prohiro.macro>.
- [10] 2023. How Many People Use WeChat? User Statistics & Trends (May 2023). <https://www.bankmycell.com/blog/number-of-wechat-users/>.
- [11] 2023. monkeyrunner. <https://developer.android.com/studio/test/monkeyrunner>.
- [12] 2023. Ranorex. <http://www.ranorex.com/mobile-automation-testing.html>.
- [13] 2023. RepetitTouch Free. <https://play.google.com/store/apps/details?id=com.cyger.repetitouch.free>.
- [14] 2023. Unity Technologies. <https://unity.com/>.
- [15] 2023. WEditor. <https://github.com/alibaba/web-editor>.
- [16] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2014. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE software* 32, 5 (2014), 53–59.
- [17] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. 2006. Surf: Speeded up robust features. In *European conference on computer vision*. Springer, 404–417.
- [18] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
- [19] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. 2020. From lost to found: Discover missing ui design semantics through recovering missing tags. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW2 (2020), 1–22.
- [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, Jiamou Sun, Sidong Feng, Zhenchang Xing, Qinghua Lu, Xiwei Xu, and Chunyang Chen. 2023. Unveiling the Tricks: Automated Detection of Dark Patterns in Mobile Applications. *arXiv preprint arXiv:2308.05898* (2023).
- [22] Sidong Feng and Chunyang Chen. 2022. GIFdroid: an automated light-weight tool for replaying visual bug reports. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 95–99.
- [23] Sidong Feng and Chunyang Chen. 2022. GIFdroid: Automated Replay of Visual Bug Reports for Android Apps. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 1045–1057.
- [24] Sidong Feng and Chunyang Chen. 2023. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. *arXiv preprint arXiv:2306.01987* (2023).
- [25] Sidong Feng, Chunyang Chen, and Zhenchang Xing. 2022. Gallery DC: Auto-created GUI component gallery for design search and knowledge discovery. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 80–84.
- [26] Sidong Feng, Chunyang Chen, and Zhenchang Xing. 2023. Video2Action: Reducing Human Interactions in Action Annotation of App Tutorial Videos. *arXiv preprint arXiv:2308.03252* (2023).
- [27] Sidong Feng, Minmin Jiang, Tingting Zhou, Yankun Zhen, and Chunyang Chen. 2022. Auto-Icon+: An Automated End-to-End Code Generation Tool for Icon Designs in UI Development. *ACM Transactions on Interactive Intelligent Systems* 12, 4 (2022), 1–26.
- [28] Sidong Feng, Suyu Ma, Jinzhong Yu, Chunyang Chen, Tingting Zhou, and Yankun Zhen. 2021. Auto-icon: An automated code generation tool for icon designs assisting in ui development. In *26th International Conference on Intelligent User Interfaces*. 59–69.
- [29] 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.
- [30] 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.
- [31] Cuiyun Gao, Wujie Zheng, Yuetang Deng, David Lo, Jichuan Zeng, Michael R Lyu, and Irwin King. 2019. Emerging app issue identification from user feedback: Experience on wechat. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 279–288.
- [32] Lorenzo Gomez, Iulian Neamtii, Tanzirul Azim, and Todd Millstein. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 72–81.
- [33] Jiaqi Guo, Shuyue Li, Jian-Guang Lou, Zijiang Yang, and Ting Liu. 2019. Sara: self-replay augmented record and replay for Android in industrial cases. In *Proceedings of the 28th acm sigsoft international symposium on software testing and analysis*. 90–100.
- [34] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. 2015. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 215–224.
- [35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
- [36] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
- [37] Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–15.
- [38] Yongjian Hu, Tanzirul Azim, and Iulian Neamtii. 2015. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 349–366.
- [39] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. PMLR, 448–456.
- [40] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. 2012. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6, 4 (2012), 1–21.
- [41] James M Keller, Michael R Gray, and James A Givens. 1985. A fuzzy k-nearest neighbor algorithm. *IEEE transactions on systems, man, and cybernetics* 4 (1985), 580–585.
- [42] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
- [43] Sotiris B Kotsiantis, Ioannis Zaharakis, P Pintelas, et al. 2007. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering* 160, 1 (2007), 3–24.
- [44] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and replay for android: Are we there yet in industrial cases?. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 854–859.
- [45] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.
- [46] Cong Li, Yanyan Jiang, and Chang Xu. 2022. Cross-device record and replay for Android apps. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 395–407.
- [47] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–163.
- [48] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldrige. 2020. Mapping natural language instructions to mobile UI action sequences. *arXiv preprint arXiv:2005.03776* (2020).
- [49] Yi Liu, Jinhui Xie, Jianbo Yang, Shiyu Guo, Yuetang Deng, Shuqing Li, Yechang Wu, and Yepang Liu. 2020. Industry practice of javascript dynamic analysis on wechat mini-programs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1189–1193.
- [50] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. 2020. WebRR: self-replay enhanced robust record/replay for web application testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1498–1508.

- [51] David G Lowe. 2004. Distinctive image features from scale-invariant keypoints. *International journal of computer vision* 60, 2 (2004), 91–110.
- [52] Daniel Müllner. 2011. Modern hierarchical, agglomerative clustering algorithms. *arXiv preprint arXiv:1109.2378* (2011).
- [53] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.
- [54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
- [55] 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.
- [56] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*. 571–582.
- [57] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An empirical analysis of UI-based flaky tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1585–1597.
- [58] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. 2011. ORB: An efficient alternative to SIFT or SURF. In *2011 International conference on computer vision*. Ieee, 2564–2571.
- [59] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.
- [60] Connor Shorten and Taghi M Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of big data* 6, 1 (2019), 1–48.
- [61] 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.
- [62] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.
- [63] Sainbayar Sukhbaatar, Joan Bruna, Manohar Paluri, Lubomir Bourdev, and Rob Fergus. 2014. Training convolutional networks with noisy labels. *arXiv preprint arXiv:1406.2080* (2014).
- [64] Tao Wang, Qingxin Xu, Xiaoning Chang, Wensheng Dou, Jiaxin Zhu, Jinhui Xie, Yuetang Deng, Jianbo Yang, Jiaheng Yang, Jun Wei, et al. 2022. Characterizing and detecting bugs in WeChat mini-programs. In *Proceedings of the 44th International Conference on Software Engineering*. 363–375.
- [65] Wei Wang and Michael W Godfrey. 2013. Detecting api usage obstacles: A study of ios and android developer questions. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 61–64.
- [66] Wenyu Wang, Wing Lam, and Tao Xie. 2021. An infrastructure approach to improving effectiveness of Android UI testing tools. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 165–176.
- [67] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.
- [68] 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.
- [69] Mulong Xie, Zhenchang Xing, Sidong Feng, Xiwei Xu, Liming Zhu, and Chunyang Chen. 2022. Psychologically-inspired, unsupervised inference of perceptual groups of GUI widgets from GUI images. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 332–343.
- [70] Jiwei Yan, Linjie Pan, Yaqi Li, Jun Yan, and Jian Zhang. 2018. Land: A user-friendly and customizable test generation tool for android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 360–363.
- [71] Shengcheng Yu, Chunrong Fang, Yexiao Yun, and Yang Feng. 2021. Layout and image recognition driving cross-platform automated mobile testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1561–1571.
- [72] Hrushikesh Zadaonkar. 2013. *Robotium automated testing for android*. Packt Publishing.

Received 2023-05-18; accepted 2023-07-31