Screencast-Based Analysis of User-Perceived GUI Responsiveness

Wei Liu[1], Linqiang Guo[1], Yi Wen Heng[1], Chenglin Li[1],Tse-Hsun (Peter) Chen[1], Ahmed E. Hassan[2]
[1]*Software PErformance, Analysis, and Reliability (SPEAR) lab, Concordia University, Montreal, Canada*
[2]*Queen's University, Canada*
w_liu201@encs.concordia.ca, g_linqia@live.concordia.ca, he_yiwen@encs.concordia.ca
chenglin.li@mail.concordia.ca, peterc@encs.concordia.ca, Ahmed@cs.queensu.ca

***Abstract*—GUI responsiveness is critical for a positive user experience in mobile applications. Even brief delays in visual feedback can frustrate users and lead to negative reviews. However, detecting and quantifying such user-perceived delays remains challenging, especially in industrial testing pipelines that evaluate thousands of apps daily across diverse devices and OS versions. Existing techniques based on static analysis or system metrics, while useful, may not accurately capture user-perceived issues or scale effectively.**

In this experience paper, we present MobileGUIPerf, a lightweight and black-box technique that measures GUI responsiveness directly from mobile screencasts—video recordings captured during automated GUI testing. MobileGUIPerf detects user interactions and visual delays, helping developers identify GUI performance issues that affect the user experience. It uses computer vision to detect user interactions and analyzes frame-level visual changes to compute two key metrics: response time (from user action to first visual feedback) and finish time (until visual feedback stabilizes). We evaluate MobileGUIPerf on a manually annotated benchmark of 2,458 interactions from 64 popular Android apps. MobileGUIPerf achieves 0.96 precision and 0.93 recall in detecting interactions, and measures response and finish times within 50 ms and 100 ms error, respectively, for over 89% of interactions. The tool has been deployed in an industrial testing pipeline and analyzes thousands of screencasts daily, uncovering responsiveness issues missed by traditional tools and improving performance debugging efficiency.

***Index Terms*—GUI responsiveness, mobile apps, mobile performance, user experiences**

I. Introduction

Mobile device manufacturers must ensure high-quality user experiences across a diverse and large set of apps. To maintain user satisfaction, they must continuously perform large-scale testing by executing automated Graphical User Interface (GUI) tests on thousands of apps daily across multiple device models and software versions. In this setting, detecting performance issues, especially those related to GUI responsiveness from the user's perspective, is both critical and technically challenging.

GUI responsiveness plays a crucial role in the user experience of mobile applications. Even brief delays in visual feedback after tapping, such as sluggish button responses, can frustrate users and make the app feel unusable. Poor responsiveness is among the top reasons users abandon apps or leave negative reviews [1], making it a critical factor in perceived app quality. However, despite its importance, accurately detecting and measuring these issues at scale remains a challenge in current industrial testing workflows.

Existing studies on mobile performance typically rely on static or dynamic analysis of source code [2, 3, 4, 5, 6, 7] or system-level metrics [8, 9, 10, 11, 12]. While these techniques provide valuable insights, they suffer from two major limitations. First, they often require access to source code or modifications to application binaries, making them difficult to apply in large-scale settings. This limitation is further exacerbated by the frequent updates of mobile apps, which demand continual instrumentation and maintenance. Second, and more importantly, these techniques do not capture performance from the user's perspective. As a result, many of the issues they detect may not be noticeable by users and have minimal impact on perceived experience [13].

To address these limitations, we collaborated with our industry partner and developed MobileGUIPerf, a lightweight and black-box technique for measuring GUI responsiveness directly from the user's perspective. Since recording video screencasts of app usage is a common practice in automated GUI testing [14], MobileGUIPerf leverages these recordings to assess responsiveness. It employs computer vision techniques to automatically 1) identify user interactions and 2) measure their responsiveness in terms of response and finish times. By analyzing screencasts, which faithfully reflect what users actually see, MobileGUIPerf enables developers to detect user-perceived performance issues.

MobileGUIPerf first applies a computer vision-based object detection model to identify visual tap indicators in screencasts, segmenting the video screencast into user interactions. These tap indicators are generated by Android's built-in `Show taps` feature, which highlights user actions such as taps by displaying a visual circle at the point of contact. MobileGUIPerf then uses frame differencing techniques based on the Structural Similarity Index Measure (SSIM) [15] to detect visual changes and compute GUI responsiveness. For each interaction, it computes two key metrics: the response time (the time from tap to the first visible change) and the finish time (the time until the visual feedback stabilizes).

To evaluate MobileGUIPerf, we constructed a benchmark dataset consisting of 2,458 user interactions from 64 of the most popular mobile apps. Each interaction was manually analyzed and annotated with the corresponding response and

finish times. Our results show that MobileGUIPerf achieves a precision of 0.96 and a recall of 0.93 in identifying user interactions from screencasts. For these interactions, MobileGUIPerf measures GUI responsiveness with high accuracy: 95% of interactions have measurement errors within 3 frames (50 ms) for response time, and 89% fall within 6 frames (100 ms) for finish time. Additionally, MobileGUIPerf is highly efficient, processing a 5-second screencast in approximately 9 seconds. The system-level recording overhead introduced by video capture is minimal–16 ms for response time and 51 ms for finish time. The tool has been integrated into our industry partner's automated testing pipeline and now analyzes thousands of recordings daily.

The main contributions of this experience paper are as follows:

- We release a manually annotated dataset containing 2,458 user interactions, each labeled with the corresponding response and finish times, to encourage future research on this important topic. The dataset is publicly available at https://anonymous.4open.science/r/gui-response-2293/.
- We present MobileGUIPerf, the first black-box technique that automatically measures GUI responsiveness from screencasts, reflecting user-perceived performance.
- MobileGUIPerf achieves high precision and recall in identifying user interactions triggered by user operations (e.g., taps or swipes), and accurately measures their GUI responsiveness in terms of response and finish times.
- MobileGUIPerf has been deployed in our industry partner's testing pipeline, where it analyzes thousands of screencasts daily and has helped uncover performance issues that traditional tools failed to detect.

Paper organization. Section II discusses background on GUI responsiveness and the limitations of existing approaches. Section III reviews related work. Section IV details our approach. Section V evaluates our approach on real-world datasets, while Section VI discusses its industrial deployment and practical impact. Section VII outlines threats to validity, and Section VIII concludes the paper.

II. Background

A. GUI Responsiveness

Responsiveness in the Graphical User Interface (GUI) is a key non-functional property that greatly impacts user satisfaction. When a mobile application responds slowly to user actions, such as tapping a button or swiping a screen, it often appears unresponsive or laggy, which can frustrate users and even lead to app abandonment or negative reviews [1, 2, 3]. Table I summarizes common GUI responsiveness metrics based on findings in human-computer interaction (HCI) and performance research [16, 17]. *Response time* refers to the time between a user action and the first visible frame update in the GUI. Prior research in HCI suggests users can feel delays in responses that take more than 100 ms [17]. Hence, even minor delays in GUI updates may thus be perceived as unresponsiveness. *Finish time* captures the duration from the user's action to when the GUI completes all visual transitions (i.e., usually when the last frame stops changing) and becomes ready for the next user interaction.

TABLE I: GUI responsiveness metrics.

GUI responsiveness	Definition
Response time	Duration from user input to the **first** visible GUI frame update.
Finish time	Duration from user input to the **final** GUI frame update.

B. Limitations of Existing Approaches

However, measuring GUI responsiveness remains challenging in practice. Most existing tools rely on system-level metrics such as CPU usage, memory consumption, or UI thread activity [11, 12, 18, 13]. These techniques, while valuable from a system standpoint, have two major limitations:

1) Limited Applicability: They often require source code access or intrusive instrumentation, which may not be feasible in black-box (e.g., testing third-party, closed-source apps for benchmarking) or large-scale testing scenarios (e.g., device manufacturers need to test hundreds of apps across many devices).
2) Lack of User Perspective: These tools monitor system-level metrics that do not always reflect what users perceive. For example, a UI thread may be busy working in the background, but does not produce any immediate visual changes on the screen [13].

As a result, developers are often left to manually inspect screencasts frame by frame—a labor-intensive process to understand how their app performs in real-world usage.

The issue becomes more challenging in large-scale testing environments. For instance, our industry partner needs to run hundreds or even thousands of automated GUI tests across many third-party apps on various mobile devices daily, making it impossible to manually diagnose performance issues through screencasts. While existing tools like NightHawk [19] and OwlEyes [20] (e.g., text overlap, misalignment) can detect visual UI issues, they primarily target display problems (e.g., text overlap and misalignment) rather than temporal aspects like UI unresponsiveness or delayed feedback. This gap highlights the need for an automated approach to assess GUI responsiveness directly from the user's visual perspective.

C. Analyzing Screencasts as a User-Centric Solution

To bridge this gap, we analyze screencasts, which are video recordings of mobile device screens that visually capture how the UI responds to user interactions. Screencasts are widely used in practice, particularly in industrial testing pipelines (e.g., for recording videos during automated testing) and bug reporting tools [14]. A screencast consists of a sequence of frames, each marked with a timestamp, providing a precise timeline of what users see. Screencasts are often recorded at the same frame rate as the device [21]. Each time the screen updates, a new frame is captured. For instance, most

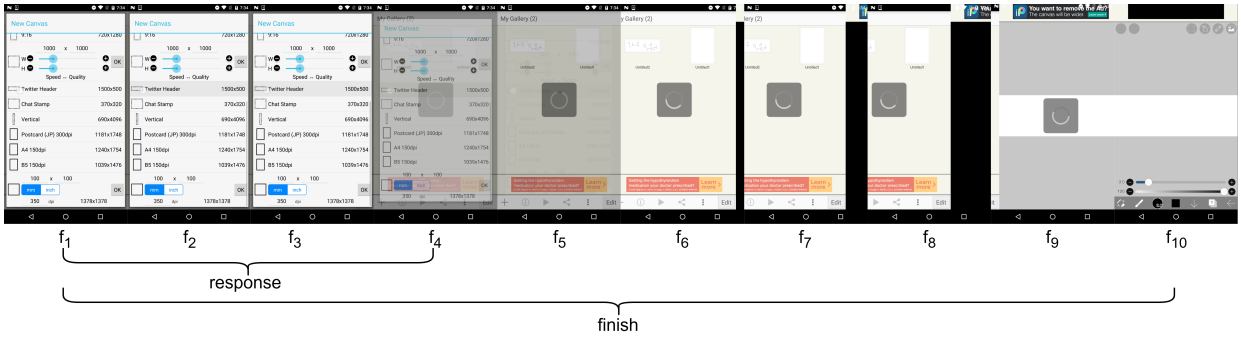

Fig. 1: A sequence of frames in a recorded screencast, with each frame annotated by index (e.g., f_1). The response duration is from f_1 to f_4 while the finish duration is from f_1 to f_{10}.

mobile devices typically operate at a standard frame rate of 60 Frames Per Second (FPS) [22], which is approximately 16.7 milliseconds [23] between consecutive frames. Each time the screen content is updated, a new frame is recorded, making screencasts a reliable source for analyzing the timing of UI responses.

Figure 1 illustrates an example screencast composed of 10 frames, each labeled with a timestamp indicating its Presentation Time Stamp (PTS). For instance, frame 1 (f_1) is shown at timestamp 0 ms, followed by frame 2 (f_2) at timestamp 16 ms, and so on. In this scenario, the user taps a UI element called "Twitter Header" in the setting screen at f_1 to create a canvas for drawing. The mobile system begins responding at frame f_4. The system then transitions to the canvas and loads the drawing tools, completing the interaction by f_{10}. As a result, the response time is calculated as f_4 - f_1, and the finish time is f_{10} - f_1.

In our collaboration with Company A, screencasts are commonly collected during automated test runs and crowdsourced bug submissions, often with the Android "Show taps" feature enabled. This option overlays a semi-transparent circle at the user contact point. As shown in Figure 2, the circle provides a visual identification of user inputs and allows developers to analyze GUI responsiveness without internal app instrumentation.

This frame-level analysis enables accurate measurement of GUI responsiveness. Since every UI update is recorded, developers can determine exactly when the system begins to react and when it stabilizes, without relying on source code access or system-level instrumentation. By analyzing screencasts, developers can automate the evaluation of GUI responsiveness and detect issues that directly impact user experience. This makes screencasts especially useful for black-box testing and performance regression analysis in continuous integration (CI) workflows.

Despite their potential, there are challenges in automatically analyzing GUI responsiveness from screencasts. Accurately detecting the start and end of visual feedback is difficult, especially with subtle transitions or partial UI updates. Noise from background animations and varied app behaviors further complicates analysis [24, 25]. Hence, we collaborate with our industry partner to address these challenges, enabling automated screencast analysis at scale.

III. Related Work

In this section, we review prior work related to our study.

Mobile Performance Analysis. Tools such as Android Lint [26], FindBugs [27], PMD [28], and Infer [29] have been widely used to detect performance issues in mobile apps. These tools typically use static analysis to identify performance anti-patterns or inefficient code structures. Prior work has extended these techniques to identify source-level performance problems [2, 30, 3, 7]. However, such tools often miss user-perceived delays and depend heavily on predefined rules, which limits their applicability in real-world scenarios.

Profiling tools such as Android Debug Bridge (adb) [11], Perfetto [12], and Android Studio Profiler [31] monitor metrics like CPU usage, memory consumption, and GPU activity. However, the performance metrics these tools monitor do not always reflect the responsiveness issues perceived by end users [13]. Dynamic instrumentation techniques (e.g., AppInsight [8], PerfProbe [9], AppSPIN [10]) collect detailed runtime events, but they still operate at the system level and may overlook visual delays. In contrast, our approach directly analyzes UI behavior through screencasts, enabling the detection of responsiveness issues from the end user's perspective.

Analysis of mobile app screencasts. Previous work has analyzed mobile screencasts to support various testing and debugging tasks, such as translating video recordings into replayable scenarios [32, 33], detecting janky frames [25], and identifying advertisements during app testing [34], and automatically replaying visual bug reports for Android apps [35]. One related technique, AdaT [24], aims to accelerate automated testing by detecting when a UI transition stabilizes. Like our method, AdaT uses computer vision techniques to classify the rendering state of frames. In contrast, MobileGUIPerf focuses on detecting user interactions and localizing keyframes to measure GUI responsiveness.

While prior techniques offer valuable insights, they often operate at the code or system level or serve different goals,

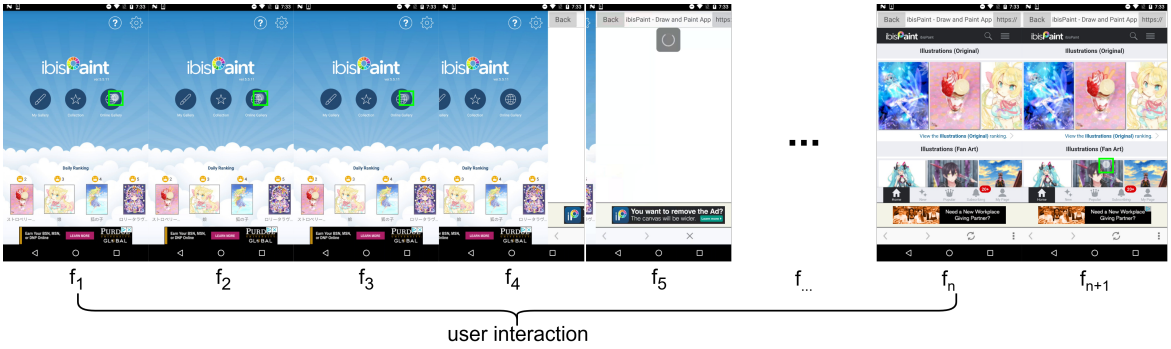

Fig. 2: Illustration of user interaction segmentation. The tap indicator (shown at f_1, f_2, f_3, and f_{n+1}) is marked as green rectangle. The first user interaction starts at frame f_1 and ends at frame f_n. The next interaction starts at frame f_{n+1}.

leaving a gap in capturing user-perceived responsiveness—precisely the focus of our work.

IV. Approach

We propose MobileGUIPerf, an automated framework developed in collaboration with Company A, to measure GUI responsiveness from the end-user perspective by analyzing mobile screencasts. MobileGUIPerf supports large-scale app testing pipelines by computing the response and finish time for each user interaction without requiring source code or instrumentation. As illustrated in Figure 3, MobileGUIPerf consists of three main components: (1) capturing video screencasts, (2) segmentation of screencast into user interactions, and (3) locating keyframes for GUI responsiveness. Algorithm 1 shows the specific steps and design details of MobileGUIPerf. The framework leverages computer vision techniques to identify user interactions and locate keyframes that indicate the response and finish of user interactions. Importantly, MobileGUIPerf operates without access to source code or instrumentation, making it suitable for black-box testing and scalable performance analysis.

A. Capturing Video Screencasts

Since video recording can introduce performance overhead, we minimize this impact by using the open-source tool scrcpy [36], which captures the device's screen via efficient video streaming. Compared to the system's built-in recorder, scrcpy introduces significantly lower CPU usage and latency, enabling a more realistic reflection of the app's actual performance. Scrcpy works by streaming the device's screen content over USB to a host machine, where the video is encoded and recorded. This process offloads the processing from the mobile device to external machines, thereby reducing runtime overhead. We record screencasts at the device's full frame rate (typically 60 FPS) to ensure that all visual changes are preserved with high temporal precision. A higher frame rate reduces the interval between consecutive frames (around 16.6 ms), capturing finer-grained visual transitions and enabling more accurate measurement of responsiveness. Once recorded, the screencasts are saved as video files for subsequent offline analysis.

Algorithm 1: MobileGUIPerf: Measuring GUI Responsiveness from Screencasts.

Input: Screencast video V recorded at f FPS
Output: List of user interactions $\mathcal{I}$ with response and finish times

Step 1: Frame Extraction
Extract all frames $\mathcal{F} = \{f_1, f_2, ..., f_n\}$ from V with timestamps.
Step 2: Tap Indicator Detection
Use Faster R-CNN to detect tap indicators in each frame.
Group frames into segments $\mathcal{S} = \{s_1, s_2, ..., s_m\}$ based on tap indicator intervals.
Step 3: User Interaction Classification
foreach *segment* $s_i \in \mathcal{S}$ **do**
    Track centroid of tap indicator across frames in s_i;
    if *indicator is static* **then**
        $type_i \leftarrow$ Tap;
    else
        $type_i \leftarrow$ Swipe;
Step 4: Keyframe Detection (Per Interaction)
foreach *segment* $s_i \in \mathcal{S}$ **do**
    Compute frame-to-frame visual similarity scores;
    Apply Isolation Forest to detect outlier frames;
    Identify response frame f_{resp} and finish frame f_{fin} based on visual change points;
    Compute:
    - Response time: $RT_i = t(f_{resp}) - t(f_{start})$
    - Finish time: $FT_i = t(f_{fin}) - t(f_{start})$

    Store interaction $\mathcal{I}_i = \langle f_{start}, RT_i, FT_i, type_i \rangle$
return $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_m\}$

B. Segmenting Screencast into User Interactions

To analyze GUI responsiveness at the level of individual actions, MobileGUIPerf segments each screencast into separate **user interactions**, i.e., sequences of frames that correspond to a single user action. This stage consists of two steps: identifying user interaction boundaries and inferring the type

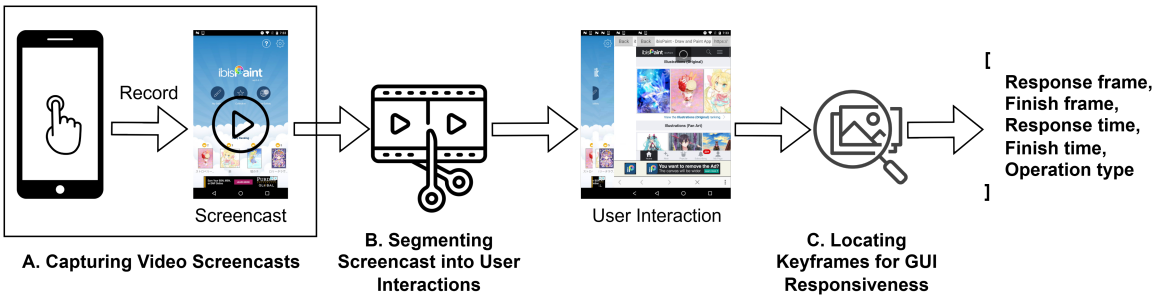

Fig. 3: The overall architecture of MobileGUIPerf.

of user action.

1) Identifying User Interaction Boundaries: We segment screencasts into user interactions by leveraging Android's `Show taps` option when recording. As shown in Figure 2, this feature overlays a semi-transparent circle at the point of contact when a user touches the screen. The circle appears under the finger, follows its movement, and typically remains visible for several consecutive frames before gradually fading. Its duration and opacity vary depending on the type of interaction (e.g., tap, swipe). While originally intended for debugging and screen recording [37, 38], this feature has been adopted in prior work on UI analysis and recording-and-replay [32, 33]. In our study, these tap indicators serve as visual cues for segmenting the screencast into individual user interactions.

We define a **user interaction** as a sequence of frames that begins with a frame showing the onset of a tap indicator and ends just before the start of the next indicator (i.e., the next user action). As illustrated in Figure 2, a tap indicator appears at frame 1, remains visible in frames 2 and 3, and disappears in frame 4. If the next tap appears at frame n+1, the frames from 1 to n are considered a single user interaction. Since the tap indicator only appears for a very short time on a few frames before the end of GUI reactions, users typically tap the screen after the previous indicator has already disappeared.

To identify interaction boundaries, we first extract all frames from the screencast and apply a pre-trained object detection model [32] based on Faster R-CNN [39] to detect the presence and position of tap indicators. This model was trained by prior work [32] on a synthetic dataset comprising 15,000 UI images, generated by superimposing touch indicators with varying opacity levels onto 5,000 unique Android app screenshots. This large-scale training corpus enables the model to generalize well across diverse mobile interfaces. Because tap indicators are often visually fused with app content, traditional image processing methods can produce suboptimal results. In contrast, the Faster R-CNN–based detector achieves robust performance across varied UI contexts. Since a single user operation produces one tap indicator that typically persists across multiple consecutive frames, we group consecutive detections into tap sequences and segment the screencast into user interactions based on the first frame in each sequence.

2) Classifying the Type of User Action: To contextualize the measured responsiveness, we classify each user interaction as either a **Tap** or a **Swipe**. This binary classification aligns with our focus on GUI responsiveness metrics: Tap interactions typically involve a brief, stationary touch, where both response time (i.e., how quickly the GUI reacts) and finish time (i.e., how long until the GUI finishes reacting) are meaningful. In contrast, Swipe interactions encompass continuous gestures—such as scrolling, swiping, or drawing—where response time is more critical, and a longer finish time may reflect expected behavior rather than performance issues.

Building on the previously segmented user interactions, we analyze the spatial movement of the tap indicator within each interaction to determine its type. The indicator appears as a semi-transparent circle that follows the user's finger throughout the interaction. As a result, its motion provides a visual proxy for the user's gesture. Specifically, we compute the center point (x, y) of the bounding box enclosing the tap indicator in the first and last frames where it appears. If the movement—measured as the Euclidean distance between these two points—is less than 10 pixels, we classify the interaction as a Tap; otherwise, it is classified as a Swipe. This threshold accounts for minor detection noise in the tap indicator's position, even during stationary touches. We empirically selected 10 pixels based on visual inspection and validation against a subset of annotated interactions.

C. Locating Keyframes for GUI Responsiveness

After segmenting the screencast into user interactions, the next step is to locate two keyframes for each interaction segment: the *response frame* (f_{resp}), where the first visual feedback appears, and the *finish frame* (f_{fin}), where the GUI becomes visually stable. We can calculate the following two metrics to measure the GUI responsiveness: Response Time as $t(f_{\text{resp}}) - t(f_{\text{start}})$ and Finish Time as $t(f_{\text{fin}}) - t(f_{\text{start}})$, where $t(f)$ denotes the timestamp of frame f, and f_{start} is the first frame of the interaction.

To identify these keyframes, we first compute visual similarity scores between consecutive frames within each user interaction. Given a segment $\mathcal{S} = \{f_1, f_2, \ldots, f_n\}$, we calculate:

$$\Delta = \{\text{sim}(f_j, f_{j+1}) \mid j = 1, 2, \ldots, n-1\},$$

where $\text{sim}(f_j, f_{j+1})$ is the structural similarity index (SSIM) between frames f_j and f_{j+1}. SSIM captures structural changes

in the image and is more robust to color or resolution differences than raw pixel comparisons [15].

We then apply the Isolation Forest algorithm [40] to the sequence Δ to identify anomalous frames that reflect significant visual transitions. Isolation Forest is an unsupervised anomaly detection technique that isolates outliers using random binary partitions, which has been shown to be efficient and accurate [40]. In our context, frames with large visual differences are flagged as anomalies. We select the first anomaly as the *response frame* and the last as the *finish frame*, corresponding to the start and finish of visible GUI feedback. To improve accuracy, we incorporate domain knowledge into the anomaly detection process. Specifically, industry experts noted that end users typically associate responsiveness with substantial visual transitions (e.g., navigating to a new screen), rather than minor effects such as a button dimming. Based on this insight, we introduce an offset in the Isolation Forest algorithm to refine the position of the response frame. This adjustment helps align the detection with user expectations of responsiveness. Finally, we extract timestamps embedded in the video file to compute response and finish times based on the identified response and finish frames.

D. A Working Example of MobileGUIPerf

To illustrate how MobileGUIPerf operates in practice, we present an end-to-end example of its output. MobileGUIPerf is deployed in production and processes thousands of screencasts daily. It enables accurate and scalable measurement of GUI responsiveness directly from screencasts, without requiring access to source code, instrumentation, or prior knowledge of the app's internal logic—making it well suited for black-box and large-scale testing. Figure 4 presents an illustrative example of the output generated by MobileGUIPerf. Due to the non-disclosure agreement, we cannot show the actual production interface, but the figure reflects a representative report format. Each report corresponds to a detected user interaction in the screencast and includes response and finish times, along with severity indicators (whether it exceeds certain thresholds). Developers can directly analyze the videos frame by frame and can use this information to understand how the interaction was triggered and assess its impact based on severity, enabling efficient prioritization and debugging.

V. Evaluation

A. Experimental Setup

To evaluate the accuracy of MobileGUIPerf in measuring GUI responsiveness, we require a dataset of screencast videos recorded during the real-world usage. While some prior studies provide mobile app recordings, existing datasets have significant limitations. For instance, the RICO [41] dataset consists of user interaction recordings in GIF format, sampled at one frame per second (i.e., 1 FPS), which is insufficient for fine-grained video analysis. Hence, we manually annotate the GUI responsiveness (i.e., response and finish time) based on the videos in the V2S dataset [32]. The dataset contains 128 videos collected from 64 popular Android applications on

Fig. 4: An illustrative interface of MobileGUIPerf.

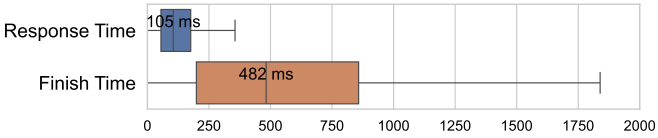

Fig. 5: Distribution of GUI responsiveness times (ms). The median response time is 105 ms, and the median finish time is 482 ms.

Google Play (top two from each of 32 app categories), such as Amazon, LinkedIn, and Airbnb. Each video was recorded with visible tap indicators overlaid on the mobile screen to illustrate user actions such as taps and swipes at a high FPS.

One limitation is that the V2S dataset does not provide frame-level annotations indicating the start, response, or finish of each user interaction. To construct a high-quality ground truth, three authors of this paper, each with over 5 years of mobile testing experience, independently annotated the video. Each annotator independently examines every frame to identify the start frame, response frame, and finish frame of every user interaction. For each user interaction, we recorded a 5-tuple: $[f_{\text{start}}, f_{\text{response}}, f_{\text{finish}}, f_{\text{end}}, \textit{type}]$, where f_{end} is computed as the last frame before the next interaction, and *type* denotes the user action type (e.g., tap or swipe). After the initial annotation, the annotators met to resolve any discrepancies and reach a consensus. These annotations were later used to evaluate the accuracy of our automated interaction detection approach.

In total, we collected 2,458 GUI-based user interactions from these 128 videos, with an average of 38 actions per mobile app. Figure 5 illustrates the distribution of response and finish times across these interactions. Overall, the median response and finish times are 105 ms and 482 ms, respectively.

RQ1: What is the Accuracy of User Interaction Detection?

A mobile screencast often records multiple user interactions performed during a GUI test. To measure responsiveness at the user interaction level, our framework first segments the screencast into individual user actions. In this RQ, we evaluate

TABLE II: Accuracy of MobileGUIPerf in identifying user interactions.

Precision	Recall	F1-Score
0.96	0.93	0.94

how accurately MobileGUIPerf can identify and segment these interactions from the screencast.

Approach. We use the manually annotated dataset (described in Section V-A) as the ground truth. In the ground truth, each user interaction is annotated as $[f_{\text{start}}, f_{\text{response}}, f_{\text{finish}}, f_{\text{end}}, type]$. In our evaluation, a user interaction detected by MobileGUIPerf is considered *correctly identified* if and only if:

1) It matches the same user operation type as the corresponding ground truth interaction (e.g., tap, swipe), and
2) It has the same start frame (f_{start}) as the ground truth annotation.

Since each user operation in the screencast corresponds to a single user interaction, both the predicted and ground truth interactions must represent only one operation.

Metrics. To evaluate the accuracy of MobileGUIPerf in identifying user interactions, we use the classic evaluation metrics: precision, recall, and F1-score. Precision is the proportion of correctly identified user interactions among all identified user interactions, recall is the proportion of correctly identified user interactions among all actual user interactions, and F1-score is the harmonic mean of precision and recall.

Results. Table II shows the results of MobileGUIPerf in identifying user interactions across mobile apps. The results are computed based on all detected and ground-truth interactions across the dataset. The tool achieves a precision of 0.96 and a recall of 0.93, demonstrating high accuracy in extracting interactions from screen recordings.

Although we achieved high precision and recall, we still observe that some user interactions are incorrectly identified. We manually examined the results and found that most of them were caused by the incorrect detection of taps. Certain UI elements may be visually similar to tap indicators, resulting in incorrect user interactions. In addition, these misclassifications may interfere with segmentation boundaries, causing valid interactions to be incorrectly split or missed altogether.

> MobileGUIPerf can detect user interactions accurately from screencasts, achieving a precision of 0.96 and a recall of 0.93.

RQ2: How Accurate is MobileGUIPerf in Measuring GUI Responsiveness?

In this RQ, we assess how accurately MobileGUIPerf measures the responsiveness (i.e., response time and finish time) of successfully identified interactions.

Approach. We use two metrics to quantify measurement accuracy:

- **Response time accuracy.** We compute the *mean absolute error* (MAE) averaged across all user interactions. For each interaction, we calculate the absolute difference between the extracted and actual response time, measured in milliseconds, and the number of frames.
- **Finish time accuracy.** Similarly, the accuracy of finish time is measured using the *mean absolute error* (MAE), defined as the absolute difference between the extracted and actual finish times, also measured in milliseconds, and the number of frames.

TABLE III: Mean absolute differences between the measured and actual *response time* of successfully identified interactions.

(a) Frame-level differences

MAE (#frames)	=0	≤1	≤2	≤3	>3
1.2	92%	93%	94%	95%	5%

(b) Millisecond-level differences

MAE (ms)	=0 ms	≤17 ms	≤33 ms	≤50 ms	>50 ms
33	92%	93%	93%	94%	6%

As mentioned above, we report the MAE for two aspects: 1) time in milliseconds and 2) video frames. We report frame-based and millisecond-based accuracy to provide a more comprehensive evaluation. Since MobileGUIPerf operates at the frame level, frame-based accuracy provides a direct view of precision, independent of the recording frame rate. In contrast, millisecond-based errors can vary depending on the frame rate of each video. Hence, presenting two metrics provides a detailed evaluation of MobileGUIPerf under different conditions.

Results. Table III presents the distribution of response time measurement accuracy by MobileGUIPerf. At the frame level, 92% of the user interactions match exactly with the ground truth (0-frame difference). Furthermore, 95% of the user interactions are within a 3-frame margin. In terms of the actual difference in time, the MAE is only 33 ms, with over 94% of the cases having a time difference of less than 50 ms. Based on practitioner feedback, the results are widely acceptable due to their high accuracy and minimal time differences, as response times less than 100 ms are often imperceptible to users [42, 43, 44, 45].

Table IV presents the distribution of finish time measurement accuracy produced by MobileGUIPerf. At the frame level, MobileGUIPerf achieves exact matches (0-frame error) for 86% of user interactions. Furthermore, 89% of interactions are measured within a 6-frame margin. Compared to response time, we see a decline in measuring the finish time. One cause of error is the presence of animations during or shortly after the user interaction. For example, some apps display animated content, such as dynamic advertisements or banners, after user interactions. These animations introduce abrupt visual changes, which our approach may mistakenly interpret as part of the interaction response, leading to a few extra frames being included in the measured finish time. However, since the updates of such animated content occur suddenly and only occasionally, they typically cause small errors (within a few frames).

We further observe that in 11% of interactions, the finish

TABLE IV: Mean absolute differences between the measured and actual *finish time* of successfully identified interactions.

(a) Frame-level differences

MAE (#frames)	=0	≤1	≤3	≤6	>6
5.2	86%	87%	87%	89%	11%

(b) Millisecond-level differences

MAE (ms)	=0 ms	≤17 ms	≤50 ms	≤100 ms	>100 ms
198	86%	86%	87%	88%	12%

time measured by MobileGUIPerf exceeds by six frames (i.e., 100 ms) or more. This is mainly due to errors in user interaction segmentation. When the next user interaction is missed in the segmentation, MobileGUIPerf mistakenly merges two interactions into one. As a result, it may select the finish frame of the second interaction as that of the first, leading to a large overestimation. Nevertheless, such cases are relatively infrequent and do not significantly impact the overall performance of MobileGUIPerf. Its high accuracy across the majority of interactions demonstrates its effectiveness and practical usages in real-world settings.

MobileGUIPerf accurately measures response time for 92% of user interactions and 95% within three frames (≈50 ms). For finish time, 86% of interactions are measured correctly and 89% fall within six frames (≈100 ms).

RQ3: What is the Accuracy of GUI Responsiveness Measurement for Apps in Different Categories?

Given the substantial variation in design and functional behavior across mobile apps, this RQ further assesses the measurement accuracy of MobileGUIPerf across 32 app categories in our dataset. For each category, we examine the proportion of user interactions for which the response/finish time is measured with zero-frame error (matches the ground truth) and within a three-frame error.

Results. Table V presents the response time and finish time measurement accuracy across 32 app categories. Here, **accuracy** refers to the proportion of interactions whose measured time falls within a specified frame error bound (e.g., ≤3 frames for response time, ≤6 frames for finish time). Across all categories, MobileGUIPerf achieves 94.4% accuracy within three frames for response time. For finish time, 85% of interactions have zero-frame error, and 88.2% fall within six frames. We sort the app categories based on the accuracy. We find that apps with more static content (e.g., those related to Books, Education, and Shopping) tend to have higher accuracy. In contrast, apps with more visual UI or animations (e.g., Video Players) tend to have a lower accuracy.

The difference in category rankings between response time and finish time is due to the different main factors that affect their measurement. For response time, lower-ranked app categories often contain reactive UI animations (e.g., a button dimming) that may be mistakenly identified as valid responses. In contrast, finish time is influenced by a wider range of factors. Since the finish frame comes after the response frame in the interaction, it is more likely to be affected by tap indicator misdetections between the actual response and finish frames. In addition, animations or video playback that continue after the finish frame can obscure the actual end of the finish. Due to these additional challenges, finish time measurement is slightly less accurate than response time measurement.

Overall, these results suggest that MobileGUIPerf performs reliably across a wide range of app categories, with high accuracy even in visually complex or animation-heavy apps. This highlights its robustness in handling diverse UI designs and interaction styles commonly found in mobile applications.

MobileGUIPerf achieves reliable accuracy across diverse app categories. Across 32 app categories, 94.4% of interactions are measured within three frames for response time, and 88.2% within six frames for finish time. Accuracy remains high even for visually complex apps, demonstrating MobileGUIPerf's broad applicability.

RQ4: How Useful is MobileGUIPerf for Performance Alerting?

In practice, developers and testers are typically less concerned with the exact response or finish time of each interaction (e.g., 50 ms vs. 55 ms), but more focused on whether it feels slow or unresponsive from the user's perspective. To approximate this perception, thresholds like 100 ms (response time) and 1,000 ms (finish time) are widely adopted in both HCI research and industry [42, 43, 44, 45]. We investigate whether MobileGUIPerf can act as a performance alerting system by identifying interactions that exceed these thresholds. This formulation aims to determine whether a user interaction is slow enough to warrant attention, as commonly required in industrial testing workflows regarding performance alerting and issue diagnosis.

Approach. We evaluate MobileGUIPerf using a threshold-based classification method to determine whether it can accurately identify user interactions that violate standard responsiveness thresholds. We conduct an end-to-end evaluation, where MobileGUIPerf first detects all user interactions and measures the response/finish time. For each interaction, we compare the response or finish time measured by MobileGUIPerf with the corresponding ground-truth value, and consider the prediction correct if both fall on the same side of the threshold (i.e., either above or below). We evaluate classification accuracy using standard metrics (precision, recall, and F1-score) under thresholds commonly adopted in practice [42, 43, 44, 45]: 100 ms for response time and 1,000 ms for finish time, where interactions above these values are likely to feel unresponsive and warrant further investigation.

Results. As shown in Table VI, MobileGUIPerf achieves a precision of 96.8% and a recall of 96.3% for the response time threshold (>100 ms). For the finish time threshold (>1000 ms), the precision and recall are 88.1% and 90.6%, respectively. We find that the results are not really impacted by the cumulative errors from the two-step process: 1) detecting the user

TABLE V: Response time and finish time measurement accuracy across 32 app categories. App categories are sorted by the proportion of interactions with response time errors ≤3 frames (left) and finish time errors ≤6 frames (right).

Response Time Accuracy (Sorted by ≤3 frames)						**Finish Time Accuracy (Sorted by ≤6 frames)**					
App Category	=0	≤3	**App Category**	=0	≤3	**App Category**	=0	≤6	**App Category**	=0	≤6
Books & Reference	100.0	100.0	Tools	94.3	94.3	Productivity	92.5	96.2	Food & Drink	84.6	88.5
Education	100.0	100.0	Social	90.4	94.2	Tools	94.3	96.2	Photography	84.5	88.1
Travel & Local	95.9	100.0	Lifestyle	93.9	93.9	Lifestyle	95.1	95.1	News & Magazines	79.3	87.9
Shopping	97.9	99.0	Entertainment	92.4	93.7	Education	92.6	94.7	Travel & Local	85.7	87.8
Events	95.1	98.8	News & Magazines	91.4	93.1	Events	91.5	93.9	Comics	84.8	87.3
Productivity	97.5	98.8	Photography	90.5	92.9	Shopping	89.6	93.8	Weather	84.8	87.0
Beauty	98.2	98.2	Art & Design	87.3	92.1	Finance	93.5	93.5	Auto & Vehicles	86.9	86.9
Communication	96.0	97.3	Sports	86.7	91.7	Maps & Navigation	93.4	93.4	Personalization	82.2	86.7
Libraries & Demo	97.3	97.3	Personalization	91.1	91.1	Parenting	92.5	92.5	Sports	81.7	86.7
Business	95.7	96.8	Medical	81.5	90.2	Communication	90.7	92.0	Libraries & Demo	79.1	82.7
Music & Audio	94.7	96.5	House & Home	86.7	90.0	Books & Reference	88.7	91.9	Social	82.7	82.7
Finance	95.3	96.3	Comics	88.6	89.9	Entertainment	89.9	91.1	Video Players & Editors	75.0	81.2
Parenting	96.2	96.2	Weather	84.8	89.1	Dating	85.3	90.7	Health & Fitness	77.0	81.1
Dating	94.7	96.0	Food & Drink	79.5	88.5	Medical	85.9	90.2	Beauty	64.3	75.0
Auto & Vehicles	93.4	95.1	Health & Fitness	85.1	87.8	Music & Audio	84.2	89.5	House & Home	70.0	75.0
Maps & Navigation	90.8	94.7	Video Players & Editors	87.5	87.5	Business	88.2	89.2	Art & Design	69.8	74.6
Avg Response Time Accuracy: 92.2% (=0-frame), 94.4% (≤3 frames)						**Avg Finish Time Accuracy:** 85.0% (=0-frame), 88.2% (≤6 frames)					

TABLE VI: Detection results of MobileGUIPerf for unresponsive interactions under fixed thresholds.

Metric	**Threshold (ms)**	**Interaction Count**	**Precision (%)**	**Recall (%)**	**F1-score (%)**
Response Time	> 100	1299	96.8	96.3	96.5
Finish Time	> 1000	508	88.1	90.6	89.3

interactions and 2) measuring response/finish time. In short, MobileGUIPerf achieves consistently high recall across both metrics, effectively capturing the majority of unresponsive interactions. This behavior aligns well with the needs of our industry partner, as we aim to identify as many potential issues as possible that require further investigation.

MobileGUIPerf enables reliable performance alerting, achieving over 96% precision and recall for detecting slow response times, and maintains strong recall (90.6%) with reasonable precision (88.1%) for finish time issues. Its high recall helps flag most unresponsive interactions, supporting early detection and industrial prioritization.

RQ5: What is the Efficiency and Recording Overhead of MobileGUIPerf?

We evaluate two practical aspects of MobileGUIPerf: (1) the efficiency of processing recorded video, since it needs to analyze thousands of videos daily, and (2) the performance overhead introduced by screen recording, as it may affect the measurement results.

Approach. 1) Video Processing Efficiency. MobileGUIPerf leverages computer vision to analyze mobile-recorded video and measure responsiveness metrics. We measured the runtime performance on a workstation equipped with an NVIDIA RTX 4090 GPU, a 16-core AMD CPU, and 64 GB of RAM. We report the average processing time per user interaction, which includes the time for segmenting videos into user interactions and measuring responsiveness metrics. Since the length of each interaction varies, we also report the minimum and maximum processing times to highlight the range.

2) Recording Overhead. Recording the screen of mobile apps may impact app responsiveness. To evaluate this, we conducted a controlled experiment using a Google Pixel 7 smartphone running Android 14, testing two configurations. For **No Recording (Baseline)**, we use an external camera to record the mobile screen with tap indicator disabled. We infer the user operations by manually observing the moment when the user's finger physically touches the screen. This setup avoids any software-based recording overhead while still allowing us to annotate response time and finish time based on visible UI changes. When using **scrcpy** (recording framework used in MobileGUIPerf), we enable screen recording with tap indicator turned on.

Since the goal of MobileGUIPerf is to measure the human-perceived GUI responsiveness, we conduct the performance measurement manually. We randomly selected three mobile apps from the top apps. For each app, we identified 10 distinct user interactions and repeated each interaction 10 times using Pixel 7. For each interaction in the recorded video, we manually annotated the response and finish times across the 10 repetitions, and computed their average to obtain a representative value. We repeat the process for the baseline and for using `scrcpy`. To enable fair comparison, we compared the same interaction across different recording settings. We computed the **Δ Response Time (ms)** and **Δ Finish Time (ms)**, i.e., the difference between each recorded configuration and the baseline, for each corresponding interaction.

Results. MobileGUIPerf processes each frame in approximately 30 ms on average, and its total processing time scales linearly with the number of frames. For instance, a 5-second user interaction recorded at 60 fps (300 frames) requires about 9.0 s to process. As also observed with our industry partner, MobileGUIPerf can process thousands of recordings within an hour. In terms of recording overhead, using `scrcpy` intro-

duces only a small average error of Δ16 ms in response time and Δ51 ms in finish time. These results suggest that `scrcpy`-based recording is efficient and adds minimal performance overhead, which is mostly unnoticeable by humans [42, 43, 44, 45], making it suitable for measuring GUI responsiveness.

> MobileGUIPerf processes each frame in $\sim$30 ms (about 9 s for a 5-second interaction at 60 fps), and introduces minimal recording overhead (Δ16 ms response, Δ51 ms finish), making it efficient and practical for analyzing GUI responsiveness.

VI. Production Deployment and Feedback from Our Industry Partner

In this section, we discuss the feedback that we received from our industry partner regarding the use of MobileGUIPerf in production testing environments.

A. Integration into Industrial Testing Pipeline

MobileGUIPerf is deployed in an industrial setting and integrated into their existing automated mobile testing pipeline. The pipeline executes thousands of automated GUI tests across hundreds of mobile apps daily. Every test is recorded in high resolution with visible tap indicators turned on. MobileGUIPerf processes these screencasts in parallel, automatically identifying user interactions, measuring response and finish times, and flagging interactions that exceed user-perceived latency thresholds. Since MobileGUIPerf required no changes to the test scripts or application binaries, the tool integration was seamless. This ease of adoption was particularly appreciated, as modifying app code or injecting instrumentation is often infeasible in large-scale testing.

B. Values in Uncovering User-Perceived Issues

A major benefit reported by the development team was MobileGUIPerf's ability to detect GUI responsiveness issues that other tools failed to identify. Conventional approaches, such as performance profilers, execution traces, and log-based analysis, rely on system-level metrics that often do not correlate with what users perceive and experience. As a result, subtle but perceptible delays were often overlooked. The feedback we received was that MobileGUIPerf, which analyzes screencasts to flag interactions that appear visually delayed, provides them a better understanding of the user-perceived performance issues. In several instances, the tool uncovered problems that had been previously flagged by users in negative app reviews, but which had not been reproducible or diagnosable using internal tools. These findings highlighted the gap between traditional performance monitoring and user-perceived responsiveness, and how approaches like MobileGUIPerf can fill this gap.

C. Actionable Feedback and Debugging Support

MobileGUIPerf also helped improve the developer experience during performance debugging. The tool outputs precise timestamps and specific start/end frames for each interaction. It also provides annotated video segments that exceed responsiveness thresholds. Developers found this output intuitive and actionable, where they can visually inspect a small video segment and immediately understand when and where the delay occurred. For instance, one developer mentioned that "Before MobileGUIPerf, we had to guess whether a delay reported by QA was real or perceptible. Now, we just click the video segment and see the problem unfold frame by frame." In summary, the positive feedback highlights the benefits of MobileGUIPerf in debugging support and how it provides a different perspective of performance debugging.

VII. Threats to Validity

External Validity. One threat to external validity is the generalizability of our evaluation dataset. While it includes 64 apps from 32 diverse categories, the dataset may not fully represent the wide range of mobile app designs and usage scenarios in practice. Future studies may evaluate MobileGUIPerf on a broader range of apps, including a wider variety of interaction styles.

Construct Validity. A major threat to construct validity is human error in the manual video annotation process in establishing the ground truth. To mitigate the risk, three experienced authors independently labeled the start, response, and finish frames for over 2,400 user interactions. Disagreements were resolved through discussion and consensus.

Internal Validity. Screencast recording and enabling the tap indicator may introduce performance overhead that could influence the internal validity of the results. However, our findings in RQ5 show that the overhead is minimal and mostly imperceptible to users.

VIII. Conclusion

In this paper, we presented MobileGUIPerf, a practical framework for measuring GUI responsiveness in mobile applications using recorded screencasts. Our evaluation shows that MobileGUIPerf achieves high precision and recall in detecting interactions and provides accurate responsiveness measurements across diverse app categories. When used as a performance alerting system, it effectively flags interactions that exceed industry-relevant thresholds. We further deployed MobileGUIPerf in an industrial setting, where it received positive feedback for uncovering user-perceived issues that existing tools failed to detect. Overall, MobileGUIPerf offers a practical and effective solution for detecting GUI unresponsiveness in mobile apps by integrating computer vision techniques into performance analysis.

References

[1] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE software*, vol. 32, no. 3, pp. 70–77, 2014.

[2] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 1013–1024.

[3] S. S. Afjehei, T.-H. Chen, and N. Tsantalis, "iperfdetector: Characterizing and detecting performance anti-patterns in ios applications," *Empirical Software Engineering*, vol. 24, pp. 3484–3513, 2019.

[4] W. Li, Y. Jiang, C. Xu, Y. Liu, X. Ma, and J. Lü, "Characterizing and detecting inefficient image displaying issues in android apps," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 355–365.

[5] T. Das, M. D. Penta, and I. Malavolta, "Characterizing the evolution of statically-detectable performance issues of android apps," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2748–2808, 2020.

[6] W. Song, M. Han, and J. Huang, "Imgdroid: Detecting image loading defects in android applications," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, 2021, p. 823–834.

[7] B. Cui, M. Wang, C. Zhang, J. Yan, J. Yan, and J. Zhang, "Detection of java basic thread misuses based on static event analysis," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1049–1060.

[8] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "Appinsight: mobile app performance monitoring in the wild," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USA: USENIX Association, 2012, p. 107–120.

[9] D. K. Hong, A. Nikravesh, Z. M. Mao, M. Ketkar, and M. Kishinevsky, "Perfprobe: A systematic, cross-layer performance diagnosis framework for mobile platforms," in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2019, pp. 50–61.

[10] Z. Lei, W. Zhao, Z. Ding, M. Xia, and Z. Qi, "Appspin: reconfiguration-based responsiveness testing and diagnosing for android apps," *Automated Software Engg.*, vol. 29, no. 2, Nov. 2022.

[11] A. Developer. Android debug bridge (adb). Accessed: February 2025. [Online]. Available: https://developer.android.com/tools/adb

[12] Google. Perfetto - system profiling, app tracing and trace analysis. Accessed: February 2025. [Online]. Available: https://perfetto.dev/

[13] J. Fu, Y. Wang, Y. Zhou, and X. Wang, "How resource utilization influences ui responsiveness of android software," *Information and Software Technology*, vol. 141, p. 106728, 2022.

[14] H. Kuramoto, D. Wang, M. Kondo, Y. Kashiwa, Y. Kamei, and N. Ubayashi, "Understanding the characteristics and the role of visual issue reports," *Empirical Softw. Engg.*, vol. 29, no. 4, jun 2024. [Online]. Available: https://doi.org/10.1007/s10664-024-10459-3

[15] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[16] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 2012, pp. 113–126.

[17] V. Roto and A. Oulasvirta, "Need for non-visual feedback with long response times in mobile hci," in *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, ser. WWW '05, 2005, p. 775–781.

[18] Google. Record a system trace. Accessed: February 2025. [Online]. Available: https://developer.android.com/studio/profile/cpu-profiler

[19] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Nighthawk: Fully automated localizing ui display issues via visual understanding," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 403–418, 2022.

[20] ——, "Owl eyes: Spotting ui display issues via visual understanding," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 398–409.

[21] S. Overflow. What is the frame rate of screen record. Accessed: February 2025. [Online]. Available: https://stackoverflow.com/questions/29546743/what-is-the-frame-rate-of-screen-record/44523688

[22] A. Developers. Frame rate. Accessed: February 2025. [Online]. Available: https://developer.android.com/media/optimize/performance/frame-rate

[23] G. for Developers. Android performance patterns: Why 60fps? Accessed: February 2025. [Online]. Available: https://www.youtube.com/watch?v=CaMTIgxCSqU

[24] S. Feng, M. Xie, and C. Chen, "Efficiency matters: Speeding up automated testing with gui rendering inference," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 906–918.

[25] W. Liu, F. Lin, L. Guo, T.-H. Chen, and A. E. Hassan, "Guiwatcher: Automatically detecting gui lags by analyzing mobile application screencasts," in *2025 IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025.

[26] A. Developers. Improve your code with lint checks. Accessed: February 2025. [Online]. Available: https://developer.android.com/studio/write/lint

[27] FindBugs. Find bugs in java programs. Accessed: February 2025. [Online]. Available: https://findbugs.sourceforge.net/

[28] PMD. Pmd an extensible cross-language static code analyzer. Accessed: February 2025. [Online]. Available: https://pmd.github.io/

[29] Infer. Infer static analyzer. Accessed: February 2025. [Online]. Available: https://fbinfer.com/

[30] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently manifesting asynchronous programming errors in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 486–497. [Online]. Available: https://doi.org/10.1145/3238147.3238170

[31] Google. Profile your app performance. Accessed: February 2025. [Online]. Available: https://developer.android.com/studio/profile#start-profiling

[32] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "Translating video recordings of mobile app usages into replayable scenarios," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 309–321.

[33] C. Bernal-Cárdenas, N. Cooper, M. Havranek, K. Moran, O. Chaparro, D. Poshyvanyk, and A. Marcus, "Translating video recordings of complex mobile app ui gestures into replayable scenarios," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1782–1803, 2023.

[34] L. Guo, W. Liu, Y. W. Heng, Tse-Hsun, Chen, and Y. Wang, "Popsweeper: Automatically detecting and resolving app-blocking pop-ups to assist automated mobile gui testing," 2024. [Online]. Available: https://arxiv.org/abs/2412.02933

[35] S. Feng and C. Chen, "Gifdroid: automated replay of visual bug reports for android apps," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1045–1057. [Online]. Available: https://doi.org/10.1145/3510003.3510048

[36] Genymobile. scrcpy: Display and control your android device. Accessed: March 2025. [Online]. Available: https://github.com/Genymobile/scrcpy

[37] A. Developers. Record a video. Accessed: February 2025. [Online]. Available: https://developer.android.com/studio/debug/am-video

[38] ——. Configure on-device developer options. Accessed: February 2025. [Online]. Available: https://developer.android.com/studio/debug/dev-options

[39] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: towards real-time object detection with region proposal networks," in *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, p. 91–99.

[40] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 413–422.

[41] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A mobile app dataset for building data-driven design applications," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 845–854.

[42] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, ser. AFIPS '68 (Fall, part I). New York, NY, USA: Association for Computing Machinery, 1968, p. 267–277.

[43] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994.

[44] D. Sillars, *High Performance Android Apps: Improve Ratings with Speed, Optimizations, and Testing*. O'Reilly Media, Inc., 2015.

[45] M. Hort, M. Kechagia, F. Sarro, and M. Harman, "A survey of performance optimization for mobile applications," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2879–2904, 2022.