Automating Visual Testing in Android Applications

Oleksii Cherkashyn
Blynk Technologies Inc., Brickell Avenue 951, 33131 Miami, FL, USA
oleksii.sciencepapers@gmail.com

Keywords: Android, Android Application Testing, Mobile Test Automation, WebdriverIO, Appium, Pixelmatch.

Abstract: Automated visual testing has become increasingly important in Android application development due to the rapid growth of device fragmentation, diverse screen resolutions, multiple operating system versions, and the widespread adoption of cross-platform frameworks. Ensuring consistent user interface (UI) rendering across different environments is a critical quality attribute, particularly for applications distributed to large and heterogeneous user bases. Traditional functional testing approaches are often insufficient to detect subtle visual regressions that directly affect user experience. Therefore, reliable and scalable visual test automation techniques are essential. This study extends existing research by empirically evaluating pixel-level visual testing across multiple execution environments. The results demonstrate that automated visual testing is feasible and stable in an Android headless emulator environment. Furthermore, the cross-platform nature of mobile applications, including different development frameworks, does not negatively impact the effectiveness of the approach. The findings also confirm that the execution platform-whether a real device, a local emulator, or a cloud-based environment-does not significantly influence test reliability. Finally, the implementation based on the Pixelmatch library shows stable and consistent performance across all experimental conditions. These results highlight the practical applicability and robustness of the proposed visual testing methodology for Android applications.

1 INTRODUCTION

Mobile applications are widely adopted, with millions of users worldwide relying on them for everyday tasks [2]. As of August 2024, approximately 1.7 million Android applications were available on the Google Play Store [1]. With the continuous growth and increasing complexity of mobile applications, ensuring their quality has become a critical challenge. In previous work [3], the importance and growing relevance of mobile application test automation were highlighted.

Mobile applications are inherently GUI-intensive and event-driven, making the Graphical User Interface (GUI) a central component through which most functionality is exposed [4]. Therefore, systematic and reliable testing of GUIs is essential to ensure both functional correctness and a high-quality user experience [5]. However, compared to web applications, mobile applications provide more limited access to UI element attributes and rely on platform-specific view hierarchies, which complicates traditional attribute-based validation approaches. To address these limitations, visual testing has emerged as an important complementary technique, enabling validation of UI elements based on their rendered appearance rather than structural properties. This is particularly relevant for elements such as images, charts, and dynamically rendered text, where correctness depends on visual representation.

This paper presents an approach to automated visual testing of Android mobile applications based on the integration of the WebdriverIO framework [7] and the Appium automation tool [6]. The implementation leverages the Node.js ecosystem, which provides a large set of reusable libraries through the Node Package Manager (npm) [8], enabling flexible and scalable test automation environments. Visual validation is achieved using a pixel-level image comparison strategy implemented with the Pixelmatch library [9] and the Chai assertion framework [10].

Unlike many existing approaches that focus on either attribute-based validation or advanced AI-driven techniques, this study investigates the feasibility and stability of pixel-level visual testing across different execution environments, including

real devices, emulators, and cloud-based platforms. In particular, special attention is given to the use of headless Android emulators, which remain underexplored in the context of visual GUI testing.

To structure the study, the following research questions are defined:

- RQ1: Can pixel-level visual testing be reliably executed on headless Android emulators?
- RQ2: How does the execution environment (real device, local emulator, cloud-based emulator) affect the stability and reproducibility of visual testing results?

The main contributions of this paper are as follows:

- C1: An implementation of a pixel-level visual testing pipeline for Android applications using WebdriverIO, Appium, and Pixelmatch;
- C2: An empirical evaluation of the proposed approach across multiple execution environments, including real devices, local emulators, headless emulators, and cloud-based platforms;
- C3: An analysis of the practical limitations of pixel-based visual testing, including dependency on device-specific baselines and challenges related to dynamic UI content.

2 RELEVANCE

Since only 8% of mobile application development projects currently leverage automated testing practices, the automation of testing represents a timely and significant area of research [12]. Visual test automation as a testing approach in mobile applications is of particular importance because, unlike web applications - where a wide range of test cases can be verified through element attributes, DOM structures, and CSS styles - mobile applications provide limited access to UI element attributes and rely on a more constrained and platform-dependent view hierarchy. As a result, many UI aspects in mobile applications cannot be reliably validated using traditional attribute-based assertions alone. Visual testing becomes essential for verifying UI components whose correctness depends on their visual appearance rather than their structural or semantic properties, such as charts, images, rendered text, graphical indicators, and validation messages. Therefore, automated visual testing serves as a critical complement to functional and attribute-based testing in mobile application quality assurance.

In their scientific work [11], Kraus, Roessler and Sulzmann describe that, based on the principles of golden master testing, they compare the actual GUI state against an expected GUI state. Accuracy is a critical factor in visual testing, as it directly affects the reliability of detected visual differences. Also, according to their scientific work, abstract GUI state (AGS) is:

- Sufficiently expressive to capture the essence of a GUA including visible and non-visible properties;
- Computationally tractable to compare two (expected vs. actual) GUI states and to identify any changes;
- Customizable to let the user decide what changes are important and which are not;
- Platform-independent to achieve a high degree of reusability.

This study is based on the concept of accuracy evaluation, in which the actual rendered image of a UI element is compared with its expected reference image. The comparison is performed with maximum precision, including pixel-by-pixel verification as well as validation of image dimensions such as width, height, and overall size.

The proposed solution is platform-independent and does not rely on a specific Android environment. It operates consistently across different execution contexts, including a local emulator running on a developer's machine, a physical mobile device, or a cloud-based emulator service. Furthermore, when executed on a local emulator, the solution remains independent of the host operating system, whether Microsoft Windows, Linux, or macOS.

This universality ensures broad applicability, reproducibility, and compatibility across heterogeneous development and testing infrastructures.

3 ANALYSIS OF CURRENT RESEARCH

In their academic work [12], Ardito, Coppola, Morisio, and Torchiano highlight that Android applications require distinct interaction patterns from users, and they emphasize that well-designed apps must deliver a high level of usability and work correctly on the broad variety of devices supported by the Android operating system. Because of these unique demands and diversity of platforms, they argue that thorough and systematic testing of the graphical user interfaces (GUIs) is essential to ensure consistent functionality and a positive user

experience. For their experiment, the authors selected two GUI testing tools representing different generations of Android testing. EyeStudio (visual testing) automates GUI interactions using image-based Capture & Replay, evolving from Sikuli and suitable for desktop-emulated apps. Espresso (layout-based testing) interacts with GUI components via properties like IDs and text, using Hamcrest matchers and JUnit for script execution, and is widely adopted in Android development; tests were run on a Nexus 5X emulator.

Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen propose Humanoid [13], an automated input generation approach for Android app testing that leverages knowledge learned from real human interaction traces. The system trains a deep learning model offline to predict the most probable user actions in a given GUI state, and then uses these predictions during online testing to guide input generation instead of relying on purely random strategies. By combining the learned interaction model with a UI transition graph, Humanoid prioritizes human-like actions and aims to reach important application states more efficiently and achieve higher test coverage.

Liu, Li, Chen, Wang, Wu, Wang, Hu, and Wang propose VisionDroid [14], a vision-driven automated GUI testing approach that leverages a Multimodal Large Language Model (MLLM) to detect non-crash functional bugs in mobile applications. Their method aligns GUI screenshots with extracted textual and structural information (e.g., widget attributes and app metadata) to enhance semantic understanding of the interface. The approach includes a function-aware explorer that guides deeper and functionality-focused exploration, as well as a logic-aware bug detector that analyzes coherent action sequences to identify inconsistencies between GUI transitions and expected processing logic. Additionally, they employ in-context learning with examples from issue reports and historical test records to further improve bug detection performance.

In their work, Ran, Li, Liu, Wang, Meng, Wu, Jin, Cui, Tang, and Xie propose VTest [15], an industrial-scale visual automated GUI testing framework for mobile applications, primarily focusing on Android and other platforms. In this study, the authors rely entirely on visual information (screenshots) rather than internal API structures, which allows their approach to handle apps with non-standard UI elements. The framework identifies UI elements from images and generates test actions based on the visual context, automatically performing interactions such as taps and swipes.

In their study, Yu, Fang, Tuo, Zhang, Chen, Chen, and Su report that, as of 2024, there are 271 vision-based GUI testing studies [4]. Overall, as the authors note, the majority of the studies focus on testing approaches based on visual analysis.

Mahmood, Mirzaei, and Malek propose EvoDroid [18], an automated Android testing approach that uses evolutionary search-based algorithms to generate UI event sequences in order to systematically explore application behavior and maximize code coverage. Their method combines Android-specific program segmentation with evolutionary test generation to efficiently guide interaction sequences through different app components and GUI states. However, the approach is not focused on visual testing or UI appearance comparison, as it does not perform screenshot-based or pixel-level visual regression analysis.

Yuanhong Lan, Yifei Lu, Zhong Li, Minxue Pan, Wenhua Yang, Tian Zhang, and Xuandong Li propose DQT (Deep Q-network Testing) [19], an approach that applies deep reinforcement learning to automatically explore Android application interfaces and generate test actions. The method encodes GUI states using graph-based representations that preserve structural and semantic UI information, improving state understanding compared to traditional tabular RL approaches. By learning from runtime interactions, the system prioritizes more informative exploration paths and increases test coverage while improving fault detection in complex Android apps.

Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su propose a model-based automated GUI testing approach (APE) [20] for Android applications that dynamically constructs and refines a GUI state model during execution. Their method continuously updates the abstraction of the application's interface using runtime exploration feedback, which allows more precise modeling compared to static GUI models. By leveraging this adaptive model refinement, the approach systematically generates test events that improve exploration coverage and increase the detection of crashes in real-world Android applications.

Overall, it can be concluded that the majority of existing studies focus on the visual verification of UI elements. This paper contributes to the advancement of visual testing by further exploring the feasibility of automated visual testing using a headless Android emulator [3], a testing approach that was previously described. In addition, this study contributes to the field of visual testing through the application of the Pixelmatch library.

4 HARDWARE AND SOFTWARE CONFIGURATION

4.1 Test Infrastructure Configuration

The experimental setup for Android application automation was based on Node.js version 20.19.4. The implementation employed the WebdriverIO framework with TypeScript integration, along with Appium and the Appium WDIO service. Additionally, Pixelmatch and several supporting libraries were incorporated to facilitate image comparison and enhance the overall testing infrastructure. The complete list of dependencies and libraries is provided in the package.json file, included in Appendix (Listing A1).

At the initial stage of the study, a detailed examination of the application's UI elements was conducted. For this purpose, the Appium Inspector tool was utilized [16], as it facilitates the identification, inspection, and visual analysis of interface elements, including their attributes and hierarchical structure within the application.

Figure 1 illustrates an example of how a UI element appears in Appium Inspector. The top part shows the element in its unselected state, while the bottom part shows it selected. In this specific case, it is a single element, and the screenshot preview of the element is clearly visible for visual inspection. In Appium Inspector, it is also possible to verify the uniqueness of the selected locator. The specific method used to define the locator - such as accessibility ID, XPath, or UIAutomator - is not critical; the key requirement is that the locator must be unique. However, for stability and execution speed, the use of an accessibility ID is recommended.

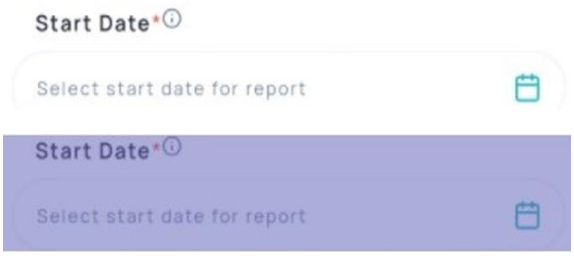

Figure 1: Visual representation of a selected and unselected element in Appium Inspector.

The next stage involves preparing the expected images of the elements. This can be accomplished using the command shown below, or alternatively, by utilizing the previously selected locator.

TypeScript code:

```
//The example uses an accessibility
ID locator.
const checkEl = await $(‘~element’);
await
checkEl.saveScreenshot('./images/ele
mScreenshot.png');
```

4.2 Preparation of the Image Comparison Library

In Appendix (Listing A2), an example of an exported function for image comparison is provided. The function utilizes the Pixelmatch library for pixel-level image comparison, pngjs for parsing PNG images, fs for file system operations, and chai for assertions. It performs a precise visual comparison between two images to verify that a mobile application's UI matches the expected design. The images are first read from files and parsed into pixel data, which includes width, height, and RGBA color values. The function checks that the dimensions of both images are identical, and if they differ, the test fails immediately. Pixel-by-pixel comparison is then performed using pixelmatch, with a small threshold to ignore minor rendering variations. A difference image is generated to visualize any discrepancies. Finally, the function asserts that no pixels differ between the actual and expected images. This approach allows for automated, exact verification of mobile UI rendering across different platforms and devices, ensuring consistency with the intended design.

Compared to a baseline usage of pixelmatch, which typically performs only raw pixel-level comparison and returns a numeric difference metric, the proposed implementation extends the core algorithm with additional layers of validation, including pre-comparison dimension checks, structured assertion handling, automatic generation of diagnostic artifacts, and full pipeline encapsulation. These enhancements improve robustness, interpretability, and usability of pixel-based visual regression testing in automated environments, particularly within continuous integration systems.

5 RESULTS AND DISCUSSIONS

This study is based on a real Samsung Galaxy A34 5G device with a screen resolution of 2340 × 1080, as well as an Android emulator configured as a Pixel 6a

device with a resolution of 2400 × 1080 and the x86_64 architecture.

The study was conducted on eight different cross-platform mobile applications, including Flutter-based, Kotlin-based, and React Native applications. A total of 179 UI element screenshots were captured and verified using automated visual testing. The dataset includes 28 images and photographs, 22 input fields, 18 dropdown menus, 25 buttons, 16 page titles, 24 text elements, and 12 icons. Additionally, 34 graphs of varying complexity and structural types were included to evaluate visual consistency across data-driven and graphical components. All tests demonstrated stable and consistent results across executions. Additionally, minor artificial modifications were introduced into the images to evaluate the sensitivity of the comparison approach. The tests successfully detected even the smallest differences, including discrepancies at the level of a single pixel (1 px), confirming the high precision of the proposed visual verification method.

The study was also conducted on the same emulator operating in headless mode using the same set of applications. The tests similarly demonstrated stable and consistent results. This experiment is particularly significant, as visual test automation studies have not previously been conducted in this mode. The findings confirm that the proposed approach remains reliable even in a headless environment, which is especially relevant for CI/CD pipelines and large-scale automated testing infrastructures. In addition, automated visual tests executed in a headless mode emulator run on average 6 - 7% faster in each test cycle compared to a UI-based emulator, which is consistent with previously conducted research findings [3].

The tests were also executed on cloud-based emulators provided by BrowserStack, as well as on locally configured Android emulators. In both environments, the automated visual tests demonstrated stable and consistent results. These findings further confirm the robustness and reproducibility of the proposed visual testing approach across different infrastructure setups, including cloud-based and local execution environments.

Each test case was executed five times on each mobile device to ensure the reliability and reproducibility of the results. The outcomes demonstrated stable and consistent performance across all environments, regardless of the Android platform, operating system version, or the cross-platform framework used in the applications. These findings confirm the robustness and platform-independence of the proposed visual testing approach.

The results also revealed several important limitations. First, the preparation of baseline (expected) screenshots is a time-consuming process. In addition, identical reference screenshots must be generated separately for each device configuration depending on screen resolution. This implies that a prepared visual test is tightly coupled to a specific device setup, regardless of whether it is executed on a real device or an emulator.

Furthermore, this approach is not suitable for automating UI elements with dynamically changing content, such as animations, video streams, or charts with continuously updating values. Since pixel-level comparison requires a static visual state, any inherent variability in the rendered content leads to test failures, even if the functionality is correct.

The evaluation of pixelmatch demonstrates several measurable advantages compared to visual regression approaches integrated in WebdriverIO via wdio-image-comparison-service and alternative JavaScript libraries including Resemble.js [17], looks-same, blink-diff, and jest-image-snapshot. In terms of execution efficiency, pixelmatch shows reduced computational overhead due to its lightweight architecture and direct manipulation of pixel buffers without framework-level abstractions, resulting in faster execution in batch image comparison scenarios compared to WebdriverIO-based and snapshot-based solutions. Table 1 presents a comparative analysis of the selected libraries. The results demonstrate that pixelmatch is the most lightweight solution among the evaluated tools. The latest version comprises 37 files across 5 directories, with a total size of 732 KB, indicating a minimal dependency footprint and potentially higher execution efficiency. Regarding determinism, pixelmatch produces fully reproducible outputs based on strict pixel-by-pixel comparison, whereas libraries such as Resemble.js and looks-same may introduce variability due to heuristic similarity metrics and configurable tolerance thresholds. In perceptual accuracy, pixelmatch leverages the YIQ color space, which improves alignment with human visual perception and reduces sensitivity to minor color deviations, in contrast to RGB-based approaches used in blink-diff that are more prone to false-positive detections. With respect to robustness, pixelmatch demonstrates improved handling of anti-aliasing artifacts, whereas looks-same and blink-diff exhibit increased noise in UI contexts involving font rendering and graphical smoothing. In terms of integration flexibility, pixelmatch operates as an

independent module without dependency on specific testing frameworks, unlike WebdriverIO visual regression services and jest-image-snapshot which are tightly coupled to their respective ecosystems. Overall, the results indicate that pixelmatch provides a combination of high performance, deterministic behavior, perceptually informed comparison, robustness to rendering artifacts, and framework independence, which makes it particularly suitable for controlled experimental setups and research-oriented visual regression analysis.

Table 1: Comparison of visual regression testing libraries.

Library	Files	Folders	Size
jest-image-snapshot	1,276	29	3.11 MB
looks-same	92	22	420 KB
blink-diff	1,108	188	9.25 MB
resemblejs	681	126	74.6 MB
pixelmatch	37	5	732 KB

6 CONCLUSIONS

Although numerous studies have been devoted to the automation of visual testing for mobile applications, the present research extends existing practices and provides additional empirical evidence. The key findings of this study are as follows:

- Automated visual testing is feasible and stable when executed on an Android headless emulator;
- The cross-platform nature of mobile applications does not affect the effectiveness of visual testing;
- The Android execution environment - whether a real device, a local emulator, or a cloud-based device - does not significantly influence the reliability of visual testing results;
- Automated mobile application testing implemented using the Pixelmatch library demonstrates stable and consistent results across different testing conditions.

REFERENCES

[1] A. Bilal, H. T. Mirza, I. Hussain, and A. Ahmad, “Investigating Influence of Google Play Application Titles on Success,” Big Data Research, vol. 36, 2024, [Online]. Available: https://doi.org/10.1016/j.bdr.2024.100443.

[2] A. Niroshan, S. Seneviratne, and A. Seneviratne, “An Empirical Study of Code Obfuscation Practices in the Google Play Store,” arXiv preprint, arXiv:2502.04636, 2025.

[3] O. Cherkashyn, “Application Test Automation in Headless Android Emulator,” Proceedings of International Conference on Applied Innovation in IT, vol. 13, no. 5, pp. 445-452, 2025, [Online]. Available: https://doi.org/10.25673/123064.

[4] Y. Wu, J. Ling, X. Luo, T. Yang, M. Zhao, C. He, and Y. Wu, “Vision-Based Mobile App GUI Testing: A Survey,” ACM Computing Surveys, vol. 58, no. 6, pp. 1-46, Oct. 2025, [Online]. Available: https://doi.org/10.1145/3773027.

[5] R. Coppola, L. Ardito, and M. Torchiano, “Fragility of layout-based and visual GUI test scripts: An assessment study on a hybrid mobile application,” in Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST ’19), New York, NY, USA, Aug. 2019, pp. 28-34, [Online]. Available: https://doi.org/10.1145/3340433.3342824.

[6] S. Adiatma and A. Darmayantie, “Implementation and Comparative Analysis of Test Automation Framework Performance for Functional Testing of Web-Based Applications using the Distance to the Ideal Alternative (DIA) Method,” Widya Teknik, vol. 22, no. 1, 2023, [Online]. Available: https://doi.org/10.33508/wt.v22i1.5027.

[7] S. Godboley, D. Dalei, R. Sadam, and D. P. Mohapatra, “Agile GUI Testing by computing novel Mobile App Coverage Using Appium Tool,” in Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, pp. 1026-1029, 2023, [Online]. Available: https://doi.org/10.1145/3555776.3577806.

[8] H. Sun, A. Rosà, D. Bonetta, and W. Binder, “Automatically Assessing and Extending Code Coverage for NPM Packages,” in 2021 IEEE/ACM International Conference on Automation of Software Test (AST 2021), pp. 40-49, 2021, [Online]. Available: https://doi.org/10.1109/AST52587.2021.00013.

[9] J. Yang, C. E. Jimenez, A. L. Zhang, K. Lieret, J. Yang, X. Wu, O. Press, N. Muennighoff, G. Synnaeve, K. R. Narasimhan, D. Yang, S. I. Wang, and O. Press, “SWE-bench Multimodal: Do AI Systems Generalize to Visual Software Domains?,” in The Thirteenth International Conference on Learning Representations (ICLR), 2025, [Online]. Availabe: https://doi.org/10.48550/arXiv.2410.03859.

[10] L. Zamprogno, B. Hall, R. Holmes, and J. M. Atlee, “Dynamic Human-in-the-Loop Assertion Generation,” IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 2337-2351, Apr. 2023, [Online]. Available: https://doi.org/10.1109/TSE.2022.3194038.

[11] L. Ardito, R. Coppola, M. Morisio, and M. Torchiano, “Espresso vs. EyeAutomate: An Experiment for the Comparison of Two Generations of Android GUI Testing,” in Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering (EASE ’19), pp. 13-22, 2019, [Online]. Available: https://doi.org/10.1145/3319008.3319022.

[12] D. Kraus, J. Roessler, and M. Sulzmann, “Visual Testing of GUIs by Abstraction,” arXiv preprint, arXiv:2007.10419, 2020, [Online]. Available: https://arxiv.org/abs/2007.10419.

[13] Y. Li, Z. Yang, Y. Guo, and X. Chen, “Humanoid: A deep learning-based approach to automated black-box Android app testing,” in Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE ’19), pp. 1070-1073, 2019, [Online]. Available: https://doi.org/10.1109/ASE.2019.00104.

[14] Z. Liu, C. Li, C. Chen, J. Wang, B. Wu, Y. Wang, J. Hu, and Q. Wang, “VisionDroid: Vision driven Automated Mobile GUI Testing via Multimodal Large Language Model,” arXiv preprint, arXiv:2407.03037v1, 2024, [Online]. Available: https://arxiv.org/abs/2407.03037v1.

[15] D. Ran, Z. Li, C. Liu, W. Wang, W. Meng, X. Wu, H. Jin, J. Cui, X. Tang, and T. Xie, “Automated Visual Testing for Mobile Apps in an Industrial Setting,” in Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP ’22), pp. 55-64, 2022, [Online]. Available: https://doi.org/10.1145/3510457.3513027.

[16] R. Mahmood, N. Mirzaei, and S. Malek, “EvoDroid: Segmented Evolutionary Testing of Android Apps,” in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE ’14), pp. 599-609, 2014, [Online]. Available: https://doi.org/10.1145/2635868.2635896.

[17] Y. Lan, Y. Lu, Z. Li, M. Pan, W. Yang, T. Zhang, and X. Li, “Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning,” in Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE ’24), art. no. 71, pp. 1-13, Feb. 2024, [Online]. Available: https://doi.org/10.1145/3597503.3623344.

[18] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, “Practical GUI Testing of Android Applications via Model Abstraction and Refinement,” in Proceedings of the 41st International Conference on Software Engineering (ICSE ’19), pp. 269-280, 2019, [Online]. Available: https://doi.org/10.1109/ICSE.2019.00042.

[19] D. S. Dupakuntla Naga, “Cross-Platform Mobile Testing with Appium: A Framework for High-Accuracy Validation in Healthcare,” International Research Journal of Engineering and Technology (IRJET), vol. 12, no. 10, pp. 450-462, Oct. 2025, [Online].

[20] A. J. Swart and P. E. Hertzog, “Quantifying the Percentage of Shading on a PV Module and Its Subsequent Impact on Its Output Power,” European Chemical Bulletin, vol. 12, special issue 3, pp. 6627-6635, Jul. 2023, [Online]. Available: https://doi.org/10.31838/ecb/2023.12.s3.747.

APPENDIX

The Appendix contains the scripts and sections of programming languages code used in the study.

```
{
  "name": "app_name",
  "type": "module",
  "devDependencies": {
    "@types/chai": "^5.0.1",
    "@types/jimp": "^0.2.28",
    "@types/moment": "^2.13.0",
    "@types/pixelmatch": "^5.2.6",
    "@types/pngjs": "^6.0.5",
    "@wdio/appium-service": "^8.39.1",
    "@wdio/cli": "^8.39.1",
    "@wdio/local-runner": "^8.39.1",
    "@wdio/mocha-framework": "^8.39.0",
    "@wdio/spec-reporter": "^8.39.0",
    "appium": "^2.11.2",
    "chai": "^5.1.0",
    "pixelmatch": "^5.3.0",
    "pngjs": "^7.0.0",
    "ts-node": "^10.9.2",
    "typescript": "^5.5.3",
    "wdio-video-reporter": "^6.1.1"
    "appium-uiautomator2-driver":
"^3.7.2"
  },
  "scripts": {
    "test": "wdio run ./wdio.conf.ts"
  },
}
```

Listing A1: Package.json.

```
 import pixelmatch from 'pixelmatch'
  import { PNG } from 'pngjs'
  import fs from 'fs'
  import { expect } from 'chai';

  export async function
  compareImages(imagePath1: any,
  imagePath2: any, expectImage: any) {
      //Read images from files
      const img1 =
  PNG.sync.read(fs.readFileSync(imageP
  ath1));
      const img2 =
  PNG.sync.read(fs.readFileSync(imageP
  ath2));

      //Checking image sizes
      expect(img1.width, `Images width
  size is different. Expexted image:
  ${expectImage}`).to.equal(img2.width
  );
      expect(img1.height, `Images
  height size is different. Expexted
  image:
  ${expectImage}`).to.equal(img2.heigh
  t);

      //Creating an object to store
  the comparison result
```

```
    const diff = new PNG({ width:
img1.width, height: img1.height });

    //Image comparison
    const numDiffPixels =
pixelmatch(img1.data, img2.data,
diff.data, img1.width, img1.height,
{ threshold: 0.1 });

    diff.pack().pipe(fs.createWriteS
tream('./images/diff/diff.png'));

    // Checking the number of
different pixels
    expect(numDiffPixels, `Images
are different. Expexted image:
${expectImage}`).to.equal(0);
}
```

Listing A2: TypeScript code.