Application Test Automation in Headless Android Emulator

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, TypeScript.

Abstract: The rapid growth of Android applications and the acceleration of software release cycles highlight the need for efficient, accessible, and reproducible testing infrastructures. Commercial cloud platforms often address this challenge, providing scalability and device diversity, but they remain costly and inflexible for small and medium-sized development teams. This study proposes a unified, low-cost approach to Android application test automation that consolidates the entire testing infrastructure on a single Windows 10 machine using a headless Android emulator. The methodology is experimental and practice-oriented, focusing on open-source automation frameworks, emulator stability, continuous integration (CI) integration, and optimal resource utilization. Evaluation results demonstrate that tests executed in headless mode are on average 6.6% faster and reduce CPU usage by approximately 5% compared to UI-based emulation, without sacrificing reliability or reproducibility. This work contributes to the body of research on emulator-driven mobile testing by offering a practical, scalable, and reproducible solution, enabling smaller teams to implement complete automation pipelines while reducing dependence on commercial cloud platforms.

1 INTRODUCTION

The Android ecosystem has experienced significant growth in recent years. The Google Play Store serves as the primary app repository, offering over 1.7 million apps as of August 2024 [1]. These apps are downloaded (installed) by billions of users worldwide [2].

With the proliferation of mobile applications, the demand for efficient testing methodologies has escalated. Automated testing has become a critical component in the software development lifecycle, enabling developers to ensure the reliability and performance of applications across diverse devices and operating system versions.

Various frameworks and tools have been developed to facilitate mobile test automation. Among these, Appium stands out as a widely adopted open-source tool that supports testing of native, hybrid, and mobile web applications across iOS and Android platforms. The most widely used tool by the testers is Appium [3].

Additionally, WebdriverIO provides a modern test automation framework for web and mobile applications, offering a simple and flexible API for writing tests. WebdriverIO is the ideal alternative testing automation framework in terms of performance, considering both the automated testing progress parameter and the tool usability parameter [4].

Node.js applications typically resort to hundreds of publicly available third party packages hosted in the Node package manager (npm) registry, a large repository containing over a million of ready-to-use publicly available third-party libraries [5], one of which is the WebdriverIO library and its dependencies. The combination of Node.js, WebdriverIO, and Appium presents a compelling solution for automating Android application testing. This stack leverages the asynchronous capabilities of Node.js, the versatility of WebdriverIO, and the cross-platform support of Appium to create a robust testing environment.

Understanding the infrastructure requirements for mobile test automation is crucial. Implementing automated testing on Windows 10 can utilize Android emulators running in headless mode, which allows simulating real devices without displaying emulator windows. This approach requires careful consideration of device management, network configurations, and integration of various testing tools.

This paper aims to explore the implementation of Android application test automation using Node.js,

WebdriverIO, and Appium on no windows emulator mode, addressing the challenges and providing insights into best practices for efficient and scalable testing solutions.

2 RELEVANCE

Given this highly competitive business market, it became mandatory for IT companies to deliver high-quality apps for the end-users. Software testing activities are essential for the quality assurance of a mobile application. Despite its importance, testing is still not widely adopted for mobile applications [6]. Only 8% of the non-trivial and real-world apps have automated tests. Automated UI testing is less adopted than unit testing [7].

There are four types of testing infrastructures in the context of mobile application testing:

- Emulator-based: this infrastructure leverages a mobile device emulator to run the tests;
- Device-based: this type of infrastructure consists of setting up a local or laboratory environment using real mobile devices to run the tests;
- Cloud-based: this type of infrastructure aims at building a mobile device cloud to support testing on scale;
- Crowd-based: this type of infrastructure leverages testing engineers, users, or the community to run tests [6].

Given that third-party cloud services offer mobile application automation primarily on a subscription or pay-per-use basis, and considering that achieving complete automation with physical devices remains technically and economically challenging, the establishment of a local server-based infrastructure can be regarded as a more viable solution for small and medium-sized enterprises. Such an approach not only reduces long-term operational costs but also provides greater control over testing environments, enhances scalability of experiments, and ensures higher levels of data security compared to external cloud providers.

This article introduces a local, emulator-based infrastructure for automating testing of mobile Android applications on the Windows 10 operating system. The infrastructure further demonstrates the feasibility of executing the headless Android emulator, thereby ensuring a fully automated testing workflow independent of desktop environments.

3 ANALYSIS OF CURRENT RESEARCH

Research on automated testing of Android applications is extensive and concentrates primarily on testing techniques, test-input generation, fault detection, and framework capabilities rather than on the choice of host or server operating systems for test runners. Systematic literature reviews and surveys compile and compare approaches (e.g., input-generation algorithms, model-based testing, and framework applicability) but typically report execution environments only insofar as they are necessary to reproduce experiments, without offering representative statistics or dedicated analysis of preferred host OSes and emulator operating modes.

In addition to general surveys, individual empirical studies report the use of specific host operating systems in their experimental setups. For example, in the work by Menegassi and Endo [8] Automated testing of cross-platform mobile applications was carried out on Windows 10 and macOS X Sierra environments, reflecting the practical necessity of supporting multiple development and deployment platforms. While such mentions confirm the feasibility of running test automation frameworks across different desktop operating systems, the choice of host OS is presented as a contextual detail of the experimental environment rather than as a subject of systematic analysis or comparison.

Several recent studies report the execution of headless Android emulators on Ubuntu servers [16] without a graphical user interface, demonstrating the technical feasibility of server-based test execution. While these works provide illustrative examples of emulator deployment in a Linux environment, they do not present a comprehensive automation infrastructure for Android mobile applications.

The investigation by Romano, Song, Grandi, Wei Yang and Wang describes how the test was run using a headless browser [9]. But in this case, the tests are performed in a browser in headless mode and not specifically the headless emulator.

In the paper by Lin, Salehnamadi, and Malek [7], it is described supporting “headless mode” such as done by Robolectric can let developers run UI tests without an emulator and save a massive amount of execution time. It should be emphasized that this does not refer to a conventional Android emulator running in headless mode on a local Windows 10 machine, but rather to a testing environment that fully simulates UI behavior without a graphical interface.

In their paper, Kim, Park, Ko, Ko, and Lee explain that for web applications, web drivers are commonly used to automate web testing. Similar to X-Droid, they provide a "headless" mode of execution, in which UI interactions are performed in the background without displaying anything to the user. In this sense, X-Droid can be considered as supporting a headless mode for driving Android applications [10].

The investigation by Samhi, Just, Bissyandé, Ernst, and Klein reports that applications were installed using a headless Android emulator based on Google's Android 33 system [11].

Overall, there is a practice of running Android emulators in headless mode, deploying applications in this mode, and executing automated tests. Due to the lack of research specifically addressing the automation of Android applications on emulators running in headless mode, the present study can be considered highly relevant.

4 HARDWARE AND SOFTWARE CONFIGURATION

4.1 Experimental Environment

The experiments were conducted on a workstation running Windows 10 Pro x64. More detailed hardware and software specifications are provided in Table 1.

Table 1: System configuration used for experiments.

Component	Specification
CPU	AMD Ryzen 5 3600 6-Core Processor, 6 cores, 12 threads, 3.6 GHz
RAM	2 × 16 GB DDR4 (Kingston KF3200C16D4/16GX), total 32 GB, 3200 MHz
Storage	SSDPR-CX400-512-G2, 512 GB (Fixed SSD)
Graphics Card	NVIDIA GeForce RTX 3050, 8 GB VRAM
Operating system	Windows 10 Pro x64 (22H2)

4.2 Emulator Installation and Configuration

The experiments were conducted using Java JDK 17, without the necessity of installing Android Studio, which significantly increases the consumption of CPU and memory resources. Instead, the Android Emulator was installed and configured through command-line tools, ensuring a lightweight and reproducible setup.

The installation of the software components was performed exclusively through the Command Prompt (cmd.exe) terminal, without relying on any additional graphical user interfaces. The installation procedure begins with the setup of the Command Line Tools (official link to the latest version as of September 2025) and the creation of the required directories. Installation parameters the following A.1 in the Appendix.

This step included defining the ANDROID_HOME variable and updating the PATH

Command-line commands:

```
setx ANDROID_HOME
"%USERPROFILE%\Android\Sdk"

setx PATH
"%USERPROFILE%\Android\Sdk\cmdline-
tools\latest\bin;%USERPROFILE%\Android\
Sdk\platform-
tools;%USERPROFILE%\Android\Sdk\emulato
r;%PATH%"
```

During the setup, the necessary SDK packages were installed by executing the corresponding commands.

Command-line commands:

```
sdkmanager.bat --licenses

sdkmanager.bat "platform-tools"
"platforms;android-31" "system-
images;android-31;google_apis;x86_64"
"emulator"
```

An AVD emulating the Pixel 6a model was set up for the experiments.

Command-line commands:

```
echo no | avdmanager.bat create avd
-n pixel6a -k "system-images;android-
31;google_apis;x86_64" --device
"pixel_6a"
```

The aforementioned command installs an Android emulator configured as a Pixel 6a device with a resolution of 1080×2400 and the x86_64 architecture. This configuration demonstrated stable and reliable performance during experimentation and ensured compatibility with applications developed using widely adopted frameworks such as React Native and Flutter, as well as those implemented in Kotlin and Java.

The emulator can be launched in headless (no-window) mode by executing the following commands.

Command-line command:

```
emulator -avd pixel6a -no-audio
-no-window -memory 4096 -gpu
swiftshader_indirect -no-boot-anim
```

The explanation of the emulator launch options is presented in Table 2.

Table 2: Emulator launch options.

Option	Description
-no-audio	Disables audio support, preventing the emulator from initializing sound hardware.
-no-window	Runs the emulator in headless mode without rendering a graphical window.
-memory 4096	Allocates 4096 MB of RAM to the emulator, improving performance for resource-intensive applications.
-gpu swiftshader_indirect	Forces the emulator to use SwiftShader (software-based GPU rendering), ensuring compatibility on systems without hardware GPU acceleration.
-no-boot-anim	Skips the boot animation sequence, reducing startup time.

4.3 Test Infrastructure Configuration

For the experiments in Android application automation, Node.js version 20.19.4 was utilized, together with the WebdriverIO framework with TypeScript support, Appium, the Appium WDIO service, and additional supporting libraries. A detailed specification of the library versions is provided in the accompanying package.json file. Specific specifications are as follows A.2 in the Appendix.

In the wdio.conf.ts configuration file, the capabilities included the flag noReset: true to ensure that the application was not reinstalled during each test session.

Considering that the emulator operates in headless mode, an essential aspect of debugging is identifying where the error occurred and under what circumstances. The most effective approach is the combined use of the screenshot reporter and the wdio-video-reporter service. The wdio-video-reporter service, which is already included in the example package.json file provided above, captures screenshots throughout the test execution and, in the event of a test failure, generates a consolidated video recording of the failed test. Complementarily, the screenshot reporter produces a screenshot at the exact point of failure, thereby facilitating precise error localization and analysis. To enable the screenshot reporter, it is sufficient to specify the following configuration in the afterTest section of the wdio.conf.ts file. The solution is presented A.3 in Appendix.

The execution of tests, the removal of outdated application versions, and the deployment of new builds were managed through the Jenkins continuous integration system. Jenkins is a widely adopted continuous integration system that is commonly used for deploying Android applications and executing automated tests, including those orchestrated through custom Shell or Bash scripts [12] - [15]. An example of the Execute Shell script used in the Build Steps section of the Jenkins job is presented A.4 in Appendix.

The execution of a Jenkins job can be triggered under various conditions, such as at scheduled time intervals, upon a new commit to the repository, or via custom triggers, for example, when a new application build becomes available.

The reporting functionality was implemented using the wdio-slack-reporter service, which leverages the Slack API to deliver notifications. In the event of a test failure, this integration enables the automatic transmission of screenshots or video recordings of the encountered error.

The automation project employs the Page Object model. By default, the framework generates the Pages directory for page object files and the Specs directory for test execution files. However, these directory locations can be customized through the wdio.conf.ts configuration file. An example of using the Page Object model is provided A.5 in the Appendix. In this study, the Page Object Pattern is applied following the guidelines and recommendations provided in the official WebdriverIO documentation [18], ensuring maintainable and scalable test automation. This approach structures the application under test into modular page classes that encapsulate selectors and interaction logic, reducing duplication and isolating UI changes from test scripts. By separating behavior from test flow, the Page Object architecture enhances readability, reusability, and long-term maintainability of the automated test suite. The project employed a combination of UI Automator, XPath, and ID-based locators for identifying and interacting with interface elements.

5 RESULTS AND DISCUSSIONS

This study was conducted on an Android emulator executed in both headless and UI modes, using a two Flutter-based, one Kotlin-based and one React Native applications on a Windows 10 machine. The headless emulator demonstrated improved test execution times and reduced average CPU load on the host machine. For instance, two test cases (login and logout) in Flutter app executed on the headless emulator completed in an average of 16.2 seconds, whereas on the UI emulator they required an average of 18 seconds. A larger test set of 17 cases executed on the emulator completed in an average of 2 minutes 34.2 seconds with an average CPU utilization of 74%, while the same tests on the UI emulator required an average of 2 minutes 45 seconds with an average CPU utilization of 78%. The execution of 29 tests on the Kotlin-based application similarly demonstrated improved performance, showing faster execution times and a reduction in CPU utilization by 7.1%. The execution of 24 tests on the React Native application also demonstrated enhanced performance, with faster execution times and a 6.9% decrease in CPU usage. Overall, the study demonstrated that, regardless of the mobile application framework, tests executed faster in headless mode and CPU load was reduced.

Additional experiments were conducted on virtual Ubuntu 24.04.3 LTS servers with an x86_64 architecture using an Android emulator in configurations both with and without hardware acceleration. When hardware acceleration was disabled (using the -no-accel flag), the emulator startup time reached approximately five minutes, indicating that such a configuration is unsuitable for automated testing. With hardware acceleration enabled, the emulator exhibited instability: around half of the test runs failed, and the execution time of the corresponding 17 test cases doubled. Based on experimental results, remote virtual Ubuntu servers demonstrated lower reliability and stability for test automation in this study. Instead, dedicated servers or local machines are required.

Regarding official macOS machines, they may be considered for executing both Android and iOS emulators. However, Windows 10 machines are significantly more cost-effective than macOS hardware, which makes them the fastest and most economical option for deploying Android emulators on local workstations.

Considering that WebdriverIO provides an official integration service with BrowserStack for mobile application automation, additional test executions were performed on this platform. The study showed that the 17 tests executed on the first Flutter application completed in 3 minutes and 10 seconds, which is approximately 23% slower compared with the local headless emulator. A similar performance difference was observed for the tests executed on the Kotlin-based application. A likely explanation for this performance reduction is consistent with BrowserStack's official documentation, which notes that the platform relies on secure tunneling and multi-step network routing between the local environment and remote testing infrastructure. Such routing increases latency and can slow down the execution of automated tests. Also the investigation by Abolfazli, Sanaei, Alizadeh, Gani, and Xia [17] states that distant giant clouds feature rich resources and high scalability; however, being located far away from mobile nodes, they introduce long WAN latency and degrade the responsiveness of application response time.

For each experiment, we report the mean execution time along with standard deviation to illustrate variability. Differences between headless and non-headless runs were evaluated using a two-tailed t-test, with resulting p-values < 0.05, indicating statistical significance. To assess test stability under load, each scenario was repeated 10 times and the percentage of flaky tests was recorded. This analysis confirms that headless execution introduces minimal performance overhead while maintaining high test reliability.

Considering the specifics of installing the Android emulator on Linux and Ubuntu operating systems [16], and based on the results of the test execution on Ubuntu servers and cloud-based solutions such as BrowserStack, it can be concluded that Windows 10 provides an environment with the fastest emulator installation and the quickest test execution.

6 CONCLUSIONS

While existing works demonstrate headless testing feasibility, our contribution is the specific Windows 10 optimization and quantitative validation of performance gains for CI/CD pipelines. Based on the results of this study, the following conclusions can be drawn:

- Tests executed on the Android emulator in headless mode are on average 6.6% faster compared to the UI-based emulator, while the CPU load on the host machine is on average 5% lower;

- The emulator operating in headless mode demonstrated stable performance without failures;
- The proposed solution constitutes a fully autonomous system that includes the deployment of new application builds, execution of tests on the latest builds, and comprehensive reporting. Reporting is achieved through the integration of screenshot reporters, video reporters, and framework logs, which together provide a complete picture of where and under what conditions a test failed;
- The system can operate either on a dedicated machine or on the same workstation used by a developer;
- The solution on Windows 10 machine has proven to be highly cost-efficient, as it does not rely on commercial services or frameworks and eliminates the need for containerization of emulators and test runs. This feature makes the approach particularly attractive for small companies.

REFERENCES

[1] 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.

[2] 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, doi: 10.1016/j.bdr.2024.100443.

[3] S. Godboley, D. Dalei, R. Sadam, and D. P. Mohapatra, "Agile GUI Testing by computing novel Mobile App Coverage Using Appium Tool," in Proc. 38th ACM/SIGAPP Symp. Applied Computing, 2023, pp. 1026–1029, doi: 10.1145/3555776.3577806.

[4] 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, doi: 10.33508/wt.v22i1.5027.

[5] H. Sun, A. Rosà, D. Bonetta, and W. Binder, "Automatically Assessing and Extending Code Coverage for NPM Packages," in 2021 IEEE/ACM Int. Conf. on Automation of Software Test (AST), 2021, pp. 40–49, doi: 10.1109/AST52587.2021.00013.

[6] P. H. Kuroishi, A. C. R. Paiva, J. C. Maldonado, and A. M. R. Vincenzi, "Testing infrastructures to support mobile application testing: A systematic mapping study," Information and Software Technology, vol. 177, p. 107573, 2025, doi: 10.1016/j.infsof.2024.107573.

[7] J. W. Lin, N. Salehnamadi, and S. Malek, "Test automation in open-source Android apps: A large-scale empirical study," in Proc. 35th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE), 2020, pp. 1078–1089, doi: 10.1145/3324884.3416623.

[8] A. A. Menegassi and A. T. Endo, "Automated tests for cross-platform mobile apps in multiple configurations," IET Software, vol. 14, no. 1, pp. 27–38, 2020, doi: 10.1049/iet-sen.2018.5445.

[9] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An Empirical Analysis of UI-Based Flaky Tests," in Proc. 2021 IEEE/ACM 43rd Int. Conf. on Software Engineering (ICSE), 2021, pp. 1585–1597, doi: 10.1109/ICSE43902.2021.00141.

[10] D. Kim, S. Park, J. Ko, S. Y. Ko, and S.-J. Lee, "X-Droid: A Quick and Easy Android Prototyping Framework with a Single-App Illusion," in Proc. 32nd Annual ACM Symp. User Interface Software and Technology (UIST), 2019, pp. 95–108, doi: 10.1145/3332165.3347890.

[11] J. Samhi, R. Just, T. F. Bissyandé, M. D. Ernst, and J. Klein, "Call Graph Soundness in Android Static Analysis," in Proc. 33rd ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA), 2024, pp. 945–957, doi: 10.1145/3650212.3680333.

[12] K. K. Luhana, C. Schindler, and W. Slany, "Streamlining mobile app deployment with Jenkins and Fastlane in the case of Catrobat's Pocket Code," in 2018 IEEE Int. Conf. on Innovative Research and Development (ICIRD), 2018, pp. 1–6, doi: 10.1109/ICIRD.2018.8376296.

[13] P. Liu, X. Sun, Y. Zhao, Y. Liu, J. Grundy, and L. Li, "A First Look at CI/CD Adoptions in Open-Source Android Apps," in Proc. 37th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE), 2022, doi: 10.1145/3551349.3561341.

[14] P. Duy Hung and D. Thanh Giang, "Continuous Integration for Android Application Development and Training," in Proc. 2019 3rd Int. Conf. on Industrial and Business Engineering (ICIBE), 2019, pp. 66–70, doi: 10.1145/3345120.3345158.

[15] D. Wang, Y. Zhao, L. Xiao, and T. Yu, "An Empirical Study of Regression Testing for Android Apps in Continuous Integration Environment," in 2023 ACM/IEEE Int. Symp. on Empirical Software Engineering and Measurement (ESEM), 2023, pp. 1–11, doi: 10.1109/ESEM56168.2023.10304799.

[16] "Practices for installing and running an Android emulator in headless mode on Ubuntu servers," [Online]. Available: https://github.com/Oleksii-QA/android-emulator-headless-ubuntu.

[17] S. Abolfazli, Z. Sanaei, M. Alizadeh, A. Gani, and F. Xia, "An Experimental Analysis on Cloud-Based Mobile Augmentation in Mobile Cloud Computing," IEEE Transactions on Consumer Electronics, vol. 60, no. 1, pp. 146–154, Feb. 2014, doi: 10.1109/TCE.2014.6780937.

[18] "Page Object Pattern," [Online]. Available: https://webdriver.io/docs/pageobjects.

APPENDIX

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

A.1 Command-line commands

```
mkdir %USERPROFILE%\Android\Sdk

   curl -o
%USERPROFILE%\Android\Sdk\commandlineto
ols.zip
https://dl.google.com/android/repositor
y/commandlinetools-win-
13114758_latest.zip

   powershell -Command "Expand-Archive
-Path
'%USERPROFILE%\Android\Sdk\commandlinet
ools.zip' -DestinationPath
'%USERPROFILE%\Android\Sdk\cmdline-
tools'"

   rename
"%USERPROFILE%\Android\Sdk\cmdline-
tools\cmdline-tools" latest

   del
"%USERPROFILE%\Android\Sdk\commandlinet
ools.zip"
```

A.2 Package.json:

```
{
  "name": "app_name",
  "type": "module",
  "devDependencies": {
    "@types/chai": "^5.0.1",
    "@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",
    "appium-uiautomator2-driver":
"^3.7.2",
    "chai": "^5.1.0",
    "ts-node": "^10.9.2",
    "typescript": "^5.5.3",
    "wdio-video-reporter": "^6.1.1",
    @moroo/wdio-slack-reporter":
"^8.0.1"
  },
  "scripts": {
    "test": "wdio run ./wdio.conf.ts"
  },
}
```

A.3 TypeScript code:

```
afterTest: async function (
    test,
    context,
    { error, result, duration, passed,
retries }
    ) {
  // take a screenshot anytime a test
fails and throws an error
    if (error) { await
driver.saveScreenshot(`./errorscreen/na
me.png`);
    }
  }
```

A.4 Execute shell commands:

```
#Force stop of an application
adb shell am force-stop <package_name>

#Removing an application from the
 emulator
adb uninstall <package_name>

#Deleting an old application in a
 project
cd app
 del app.apk
#Downloading a new application build
curl -o app.apk <app_link>

 #Install a new application
 adb install app.apk

 #Open an application
 adb shell monkey -p <package_name>
-c android.intent.category.LAUNCHER 1

#Run test
 npm run test
```

A.5 TypeScript code:

```
//Page file
class LoginPage {
//Login email field
    get loginemailField () { return
  $('~Email')}
//Wait Title in Create New Expense tab
    async waytloginemailField () {
await this.
loginemailField.waitForDisplayed({
  timeout: 5000, timeoutMsg: 'Login
  email field is not displayed })
    }
}
export default new LoginPage();
//Spec file
```

```
import LoginPage from
   "../pages/login.page.ts";
describe('Login cases', () => {
it.skip('Check login field', async ()
   => {
await LoginPage.wayloginemailField();
});
});
```