

Comparative analysis of Espresso and Appium frameworks for automated UI testing of Android mobile applications

Jakub Derkacz*

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

This paper presents an empirical comparison of two leading frameworks for automated UI testing of Android applications: Espresso and Appium. An identical suite of 65 test cases was implemented in both frameworks for the MediTrack medication management application and executed in ten independent runs. Analysis of stable runs showed that Espresso completed the test suite in 437.4 s on average, while Appium required 727.7 s, a slowdown factor of 1.66. Per-module ratios ranged from 0.71 to 2.78, depending on test complexity. Appium achieved higher run stability (80% vs 60%), while standard deviations of stable runs were comparable. The results provide empirical guidance for choosing between the frameworks based on project requirements.

Keywords: test automation; Espresso; Appium; mobile testing; comparative analysis

*Corresponding author

Email address: s95389@pollub.edu.pl (J. Derkacz)

Published under Creative Common License (CC BY 4.0 Int.)

1. Introduction

The widespread adoption of mobile applications has made software quality a critical concern for development teams. According to StatCounter [1], the Android platform holds over 71% of the global mobile operating system market, which places substantial responsibility on developers regarding the reliability of their products. Errors in mobile applications can directly affect users' finances, health, or personal data security, making systematic verification a business-critical activity rather than an optional engineering practice.

Software quality assurance is a complex process requiring a systematic approach based on verification and validation. The IEEE 829 standard defines software testing as a process aimed at detecting defects through controlled execution of the system under test [2]. Myers et al. [3] emphasised that testing is fundamentally a destructive activity — its purpose is to reveal flaws rather than confirm correctness — and this perspective has shaped modern testing methodologies.

Automated testing has become an essential practice in contemporary mobile software engineering, as manual execution of repetitive test scenarios is expensive, error-prone, and does not scale within continuous integration and continuous delivery (CI/CD) pipelines [4]. The test pyramid concept, formulated by Cohn [5] and developed further by Fowler [6], recommends that UI tests should constitute the smallest portion of an automated test suite, while unit tests should form its foundation. Nonetheless, UI tests remain essential because they verify end-to-end behaviour from the user's perspective, which no other test level can replicate fully.

Among the available tools for automated UI testing on Android, two frameworks have gained particular prominence: Espresso and Appium [7]. Espresso, officially supported by Google, operates within the testable application's process and is dedicated exclusively to the Android platform. Its declarative API

and built-in synchronisation mechanism, based on the `IdlingResource` abstraction, allow developers to write concise tests that automatically wait for the UI to reach an idle state before performing assertions [8]. Appium, in contrast, is a cross-platform solution based on the WebDriver protocol, allowing tests to be written in any language with an available WebDriver client and executed against both Android and iOS applications [9, 10]. Appium supports native, web, and hybrid applications and does not require modification of the application under test, which enables black-box testing of third-party applications.

Despite the broad industry adoption of both tools, comparative studies based on systematic empirical measurements remain relatively scarce. Existing comparisons often focus on qualitative criteria such as documentation, supported platforms, or API complexity [11, 12], while quantitative execution-time measurements under controlled conditions are less frequently reported. Decisions about which framework to adopt are therefore often made based on team experience or developer-oriented resources rather than reproducible empirical measurements.

This paper addresses this gap by presenting an empirical comparison of both frameworks under controlled conditions. Identical test suites of 65 test cases were implemented for the MediTrack medication management application — a Java-based Android application using the MVVM architecture pattern, Clean Architecture principles [13], and Firebase as the backend service. Each suite was executed in ten independent runs, yielding a total of 1,300 test case executions, which provided data sufficient for statistical analysis of execution times and stability of results.

The contribution of this work is threefold. First, it provides quantitative measurements of execution-time differences between Espresso and Appium under controlled experimental conditions. Second, it analyses

the variability of these differences across functional modules of the application, identifying scenarios in which each framework performs best. Third, it formulates practical recommendations for framework selection based on project requirements and team capabilities.

The remainder of this paper is organised as follows. Section 2 reviews related work on Android UI testing frameworks. Section 3 describes the experimental methodology, including the test application, suite structure, and measurement procedure. Section 4 presents and discusses the results. Section 5 concludes the paper with recommendations and outlines directions for future research.

2. Related work

The landscape of Android UI testing frameworks has been characterised in several comparative and survey studies. Braun et al. [7] surveyed the available mobile test automation tools and proposed a structured approach to tool selection, emphasising the difficulty teams face in identifying an appropriate solution given the breadth of available options. Da Silva and de Souza Santos [11] compared mobile testing tools using documentary analysis across multiple criteria, including supported platforms, automation capabilities, and CI/CD integration; their study confirms that no single tool dominates across all evaluation dimensions.

Zelenchuk [12] provided a practitioner-oriented guide to the Espresso framework, documenting its API, synchronisation model, and recommended patterns for writing maintainable automated UI tests on Android. Arif and Ali [14] conducted a comparative study of mobile application testing tools, identifying execution speed, cross-platform support, and ease of configuration as the most relevant evaluation dimensions for industrial practice. Lenka et al. [15] analysed the performance characteristics of automated testing tools, showing that tool selection has a measurable impact on execution efficiency depending on the type of tests performed.

Gu and Rojas [16] analysed scripted GUI testing in open-source Android projects and found that Espresso is significantly more frequently used than UIAutomator. They also reported that Espresso's simpler API and automatic synchronisation mechanism are among the reasons developers prefer it, particularly during the early phases of test suite development. Coppola et al. [17] examined the maintenance overhead of Espresso test suites, finding that 55% of modified test methods were affected by GUI-related changes, which underscores the importance of locator selection and the application of unified, cross-platform design approaches to test organisation [18]. In a subsequent study, Coppola et al. [19] also addressed the translation of widget-based tests into visual tests, identifying the trade-offs between the two approaches.

Studies focused specifically on Appium include Wang and Wu [10], who demonstrated its applicability to native, web, and hybrid mobile applications and described the configuration of the Appium server for test

automation in an emulator-based environment, and Lovreto et al. [20], who applied Appium to mobile game testing — demonstrating its black-box capabilities but also identifying limitations related to non-standard view hierarchies. Vajak et al. [21] developed an automated testing environment based on Appium for functional testing of mobile applications and identified its client-server architecture as the principal advantage for cross-platform testing. Mojahed et al. [22] reported the use of an Appium-based platform (ODACE) for certification testing of mobile devices in telecommunications scenarios, with a reported 89% reduction in human involvement compared to manual procedures.

Research on Espresso specifically has focused on its automatic synchronisation mechanism and the reliability of generated tests. Arcuschin et al. [23] studied the reliability of automatically generated Espresso tests and demonstrated that test generation techniques can produce effective regression oracles for Android applications, while highlighting the importance of unambiguous selectors for test stability. Negara et al. [24] described the Espresso Test Recorder tool, which generates instrumented tests from recorded user interactions and lowers the entry barrier for adopting test automation. Adamsen et al. [25] addressed the broader problem of systematic execution of Android test suites under adverse conditions, demonstrating that environmental disturbances such as low battery, lost network connectivity, or incoming notifications can cause test failures unrelated to the application logic.

The general issue of flaky tests — tests that produce inconsistent results under identical execution conditions — has been studied extensively by Luo et al. [26], who classified the principal causes of flakiness in industrial test suites. This phenomenon is particularly relevant to UI tests, which depend on operating system timing, network latency, and emulator load.

Lin et al. [27] analysed the accuracy and efficiency of automated test oracles for Android, providing one of the first quantitative reports on the overhead introduced by WebDriver-based automation frameworks. Their findings indicate that each WebDriver command introduces network latency on the order of hundreds of milliseconds, which is consistent with the architectural overhead observed in the present study.

Comparative analyses of mobile test automation infrastructures have also examined systematic and capture-based alternatives. Tramontana et al. [28] conducted a systematic mapping study of automated functional testing of mobile applications, identifying the principal categories of tools and approaches available and highlighting the lack of empirical comparisons under controlled conditions. Di Martino et al. [29] compared capture-and-replay techniques against automatic input generation for Android GUI testing, showing that the effectiveness of such approaches depends on the exploration strategy used and that neither approach dominates across all evaluation criteria.

In summary, the related literature confirms the relevance of comparing Espresso and Appium under controlled conditions but does not provide systematic measurements of execution-time differences across functional modules in a unified experimental setup. The present work addresses this gap.

3. Methodology

3.1. Experimental design

The experiment was designed according to the principle of controlled comparison, with the aim of minimising the influence of external factors on the results. Five design principles were adopted: identical test cases for both frameworks; multiple test runs to assess result stability; identical execution environment for both frameworks; identical test data; and consistent application of the Page Object Model pattern in both implementations.

3.2. Test application

The MediTrack application served as the experimental subject. It is a medication management system for Android that enables users to add medications, schedule dose times, track dose intake history, and review adherence statistics. The application was implemented in Java using the Model-View-ViewModel (MVVM) architectural pattern combined with Clean Architecture principles [13]. It uses Firebase Authentication for user authentication and Cloud Firestore as the cloud database, with the Room library serving as a local cache. Hilt is used for dependency injection, and the Navigation Component manages inter-fragment navigation.

The application consists of six functional modules: user login, dashboard with daily statistics, medication list, medication addition form, intake history, and user profile. This structure provides a representative range of UI test scenarios, from simple element visibility checks to multi-step navigation flows requiring authentication and data manipulation. The choice of a real-world application rather than a synthetic benchmark ensures that the measured characteristics reflect conditions encountered in industrial software projects.

3.3. Test suite design

For each framework, an identical set of 65 test cases was implemented, distributed across the six functional modules as shown in Table 1. The test cases were designed to cover three categories: visibility tests (verifying that UI components appear in the expected state), navigation tests (verifying transitions between screens), and validation tests (verifying error messages in response to incorrect input).

Table 1: Distribution of test cases across application modules

Module	Test cases
Login	11
Dashboard	12
Medication list	6
Add medication	14
History	9
User profile	13
Total	65

Both test suites were implemented using the Page Object Model pattern [18], ensuring comparable code structure and maintainability characteristics. Each application screen was encapsulated as a separate Page Object class containing the locators and operations available on that screen. Test classes then composed these Page Objects to express test scenarios at an abstraction level above the framework-specific API.

For Espresso, locators are based directly on Android View IDs (R.id.*) defined in the application's resources. For Appium, locators were defined using either resource IDs in the form "com.meditrack:id/element_id" or UIAutomator selectors for elements that could not be located reliably through the AppiumFieldDecorator mechanism. In particular, the bottom navigation tabs required the use of explicit UIAutomator selectors due to a limitation observed during initial implementation.

3.4. Experimental setup

The experiment was conducted on a workstation equipped with an Intel Core i7-13700K processor (3.40 GHz, 13th generation), 32 GB of DDR5 RAM, and a Lexar NM800PRO 2 TB SSD, running Windows 11 (64-bit). The Android emulator was configured as a virtual Pixel 8 Pro device with Android 14 (API level 34), 4 GB of allocated memory, and x86_64 processor architecture. Development tools included Android Studio (Meerkat release) and the Android SDK with platform tools version 34.

For the Appium framework, Appium server version 3.3.1 with the UIAutomator2 driver was used, together with the Java client library 'io.appium:java-client' version 8.6.0. Tests were executed using Apache Maven 3.9.9, and the Allure framework for Java was used to generate test reports. The Espresso environment used 'androidx.test.espresso:espresso-core' version 3.6.1, 'androidx.test.ext:junit' version 1.2.1, 'androidx.test.runner' version 1.6.1, and 'com.google.dagger:hilt-android-testing' version 2.51.1, with a custom test runner extending AndroidJUnitRunner to accommodate Hilt's dependency injection in the test context.

Both test suites were executed against the Firebase Emulator Suite [30] running locally on the host machine: the Authentication emulator on port 9099 and the Firestore emulator on port 8081. This configuration eliminated dependency on external services and ensured deterministic data state at the beginning of each run. The application was configured to connect to the local IP address 10.0.2.2, which the Android emulator network maps to the host machine.

3.5. Measurement procedure

Each test suite was executed ten times, yielding a total of 1,300 test case executions (650 per framework). Before each run, the emulator was restarted to ensure a clean execution state, and the Firestore database was wiped to remove any medications added by previous runs. The test user account (test@meditrack.com) was created in the

Firebase Authentication emulator before the first run and remained in place for the duration of the experiment.

For Appium runs, the procedure was as follows: the Firebase emulators were started, the Appium server was verified to be running, and the test suite was executed with the command `mvn test -Dtarget=emulator`. For Espresso runs, the test suite was executed through the Android Studio interface, and the results in XML format were collected from the `app/build/outputs/androidTest-results/connected/` directory. In both cases, the results were imported into Allure Report for analysis.

Test execution times and outcomes were recorded automatically by Allure Report, which integrates with JUnit 4 and captures detailed metadata for each test case, including execution duration, status, and stack traces in case of failure. The results were exported to CSV files for further statistical analysis. The following metrics were computed: total suite execution time per run, mean execution time per test case, standard deviation across runs, and per-module execution time aggregates. Statistical analysis was performed in Python using the standard library `statistics` module.

3.6. Threats to validity

Several threats to the validity of the experimental results were considered. Internal validity is supported by the use of an identical execution environment, identical test data, and identical test scenarios for both frameworks. External validity is limited by the use of a single hardware configuration and a single application; results may differ on physical devices or for applications with different architectural patterns. Construct validity is supported by the use of standard measurement tools (Allure Report) and direct measurement of execution time as defined by the framework's own timing instrumentation. Conclusion validity is limited by the relatively small sample size of ten runs per framework, which precludes strong inferential statistics; however, the observed effect sizes are sufficiently large to support the practical conclusions drawn in Section 4.

3.7. Reproducibility considerations

To ensure reproducibility of the experiment, all configuration parameters were documented and remained constant throughout the measurement period. The same emulator image, the same versions of all frameworks and libraries, and the same network configuration were used across all 20 runs. Source code of both test suites is organised as a single Git repository, with the Espresso suite located in the main Android project's `androidTest` directory and the Appium suite as a separate Maven module. The repository structure follows the conventions recommended by the Android documentation [13] and the unified automation structuring approach described by Purushothaman et al. [18], which simplifies replication of the experiment on different hardware or with different applications.

The Firebase Emulator Suite was used in its default configuration with project-specific service settings stored in the `firebase.json` file in the repository root. The

Firestore emulator runs on port 8081 (default 8080 was changed due to a conflict with another local service); this port number is encoded both in `firebase.json` and in the application's Hilt-managed `FirebaseModule`. This configuration enables deterministic state at the beginning of each run and avoids any dependency on a network connection during test execution.

4. Results and discussion

4.1. Suite stability

Across the ten runs of each framework, transient failures occurred in some runs due to environmental conditions (timeouts caused by emulator load), rather than logical errors in the test implementations. This observation is consistent with prior work on test flakiness [26] and on the systematic execution of Android test suites under adverse conditions [25]. Espresso recorded a total of 5 failed test cases across 4 runs (runs 3, 8, 9, and 10), while Appium recorded 2 failed test cases across 2 runs (runs 5 and 8). In all cases, repeated execution of the same test cases in subsequent runs yielded successful results, confirming the environmental nature of the failures.

For the purpose of execution-time analysis, runs that completed without any failed test cases were classified as stable. Six out of ten Espresso runs (60%) and eight out of ten Appium runs (80%) met this criterion. The remaining analysis focuses on stable runs to eliminate the influence of timeout-related artefacts on execution times. The higher stability rate of Appium was unexpected, but its origin remains unclear given the relatively small sample size; further investigation would be required to determine whether this difference is systematic or reflects specific conditions of the test environment during the experiment.

4.2. Overall execution times

Table 2 summarises the key timing statistics for both frameworks, computed for all runs and for stable runs only.

Table 2: Aggregate execution-time statistics

Measure	Esp. (all)	App. (all)	Esp. (stable)	App. (stable)
Mean [s]	593.6	745.9	437.4	727.7
Median [s]	528.6	815.4	429.1	734.3
Standard deviation [s]	291.6	103.7	108.5	109.1
Minimum [s]	333.4	586.8	333.4	586.8
Maximum [s]	1062.2	833.0	550.5	828.9

When restricted to stable runs, the mean execution time of the Espresso suite was 437.4 s, while the Appium suite required 727.7 s on average. This corresponds to a slowdown factor of 1.66 for Appium relative to Espresso, meaning that Espresso completed the identical test set approximately 40% faster. This advantage is consistent with the architectural difference between the two frameworks: Espresso operates within the application's process, whereas Appium operates through a

client-server model where each test command requires a full HTTP request-response cycle [27].

The standard deviation across all runs is markedly higher for Espresso (291.6 s) than for Appium (103.7 s), reflecting the greater impact of timeout-related failures on the Espresso suite. However, when restricted to stable runs, the standard deviations of both frameworks become comparable (108.5 s and 109.1 s, respectively), suggesting that under normal operating conditions both frameworks exhibit similar run-to-run variability. This is an important practical observation: when the test environment behaves as expected, the predictability of execution time is comparable for both frameworks.

4.3. Per-module execution times

Table 3 presents the execution-time breakdown by functional module, computed from stable runs. Two metrics are reported for each module: the total execution time of all tests in the module across one suite run, and the average execution time of a single test case within the module.

Table 3: Per-module execution times (stable runs)

Module	Esp. sum [s]	Esp. avg [ms]	App. sum [s]	App. avg [ms]	Ratio
Login	90.2	8,201	126.3	11,479	1.40
Dashboard	61.8	5,149	44.1	3,677	0.71
Medication list	39.5	6,579	105.6	17,593	2.67
Add medication	116.0	8,288	167.7	11,976	1.44
History	48.2	5,360	56.7	6,296	1.17
User profile	81.7	5,360	227.5	17,498	2.78
Total	437.4	5,360	727.7	10,888	1.66

The per-module analysis reveals significant variability in the relative performance of the two frameworks. The only module in which Appium outperformed Espresso is the Dashboard module (ratio 0.71), which consists exclusively of visibility tests for static UI elements without complex navigation. In this scenario, the initialisation overhead of Espresso (Hilt context setup, test thread synchronisation) becomes proportionally larger than the per-command network latency of Appium for simple operations. This finding indicates that Espresso's advantage is not universal but depends on test complexity — for trivial visibility checks, Appium's lower setup overhead can yield better results.

The largest disparities favouring Espresso were observed in the User Profile (ratio 2.78) and Medication List (ratio 2.67) modules. Both modules contain test cases that require multiple sequential transitions between screens — navigating to the profile, clicking a settings row, verifying the destination screen, and returning. Each such transition incurs a full HTTP request-response cycle in Appium, which is consistent with the architectural observation by Lin et al. [27] that WebDriver-based frameworks introduce overhead on the order of hundreds of milliseconds per command. The Medication List module additionally includes interactions with the floating action button and navigation to the

add-medication form, further amplifying the cumulative network overhead.

In the remaining modules (Login, Add medication, History) the slowdown factor ranged between 1.17 and 1.44, reflecting moderate amounts of UI interaction. The Login module has a relatively higher ratio (1.40) due to the cost of remote authentication request handling, where each Appium login test includes the entire form submission cycle, while Espresso benefits from in-process execution of the authentication flow.

4.4. Extreme values

The five slowest test cases across stable runs were all from the Appium suite, all related to multi-step navigation in the Profile and Medication List modules, with execution times ranging from 18,035 ms to 21,135 ms. The slowest single test ("Profile — logout navigates to login") required an average of 21,135 ms in Appium, compared to under 8,000 ms in Espresso. The five fastest test cases were also from the Appium suite, but from the Dashboard module, with execution times between 2,371 ms and 2,540 ms — faster than the corresponding Espresso tests of the same module, which ranged from 4,528 ms to 4,658 ms.

This polarisation confirms the observation that Appium's architectural overhead is most pronounced in tests with many sequential network operations and least pronounced in simple visibility checks. The implication for test suite design is that the relative cost of Appium can be reduced by minimising unnecessary navigation steps and by consolidating multiple assertions into single test cases where possible.

4.5. Implementation effort

A qualitative analysis of the implementation effort was also conducted. The Espresso environment required only the addition of dependencies to the existing Android project's build.gradle file. Use of the Hilt dependency injection library [13] necessitated a custom test runner implementation, but this was a one-time configuration cost. The Appium environment required installation of the Node.js runtime, the Appium server with the UIAutomator2 driver, and the creation of a separate Maven project alongside the main Android project. Managing several external processes — the Firebase emulators, the Appium server, and the Android emulator — in the correct order before each test session adds operational complexity not present in Espresso.

Several technical challenges were encountered during Appium implementation. First, locating bottom navigation elements required replacing the AppiumFieldDecorator with direct UIAutomator selectors, as the decorator failed to locate elements identified solely by resource ID in the bottom navigation component. Second, designing a reliable return-to-home mechanism required handling multiple possible application states, as the application could be left in different fragments after previous tests. Third, locating form validation error messages required analysis of the view hierarchy via the Appium Inspector tool, as the error

text was rendered in a separate `TextView` with a non-obvious resource identifier.

Espresso's declarative API allowed concise test code: a single visibility check is expressed in one statement (`onView(withId(R.id.x)).check(matches(isDisplayed()))`). The equivalent Appium test requires explicit handling of waits and asynchronicity, typically resulting in two to three times as many lines of code per test case. This observation is consistent with the findings of Gu and Rojas [16], who reported that Espresso's simpler API contributes to its higher adoption rate in open-source projects.

4.6. Discussion

The empirical results support several practical observations about the trade-offs between Espresso and Appium. The 1.66-fold execution-time advantage of Espresso is substantial and is most pronounced in test scenarios involving complex navigation flows. For projects with frequent test executions in CI/CD pipelines [4], this difference accumulates to significant time savings over time. For example, a daily nightly build with both test suites would save approximately five minutes per execution, which translates to over thirty hours of CPU time per year.

Appium's higher stability rate (80% vs 60%) was an unexpected finding, although the small sample size of ten runs per framework precludes strong conclusions about systematic differences in stability. The comparable standard deviations in stable runs suggest that both frameworks exhibit similar intrinsic variability when environmental factors are controlled.

The results are consistent with prior comparative studies [11-15] in confirming that no single framework dominates across all evaluation criteria. The choice between Espresso and Appium should therefore be guided by project-specific factors: cross-platform requirements, code access, team expertise, and the relative importance of execution speed versus implementation flexibility.

Espresso is the preferred choice when the development team maintains the tests, has access to the application source code, and targets only the Android platform. In such cases, the framework's tight integration with Android Studio, its declarative API, and its execution speed make it the most productive option [16, 24]. Appium remains the preferred choice for cross-platform projects [21], for black-box testing of applications without source code access [22], or in organisations with existing WebDriver-based testing infrastructure (e.g. Selenium for web testing). For applications targeting both Android and iOS, Appium is effectively the only practical choice for maintaining a single test suite across both platforms.

4.7. Practical implications for CI/CD pipelines

The execution-time difference between Espresso and Appium has direct consequences for the design of continuous integration pipelines. In a hypothetical project performing twenty test suite executions per day

(e.g. pull request validation, scheduled nightly builds, and on-demand runs), Espresso would require approximately 2.4 hours of cumulative execution time per day, while Appium would require approximately 4.0 hours. Over a working year of 250 days, this corresponds to a difference of about 400 hours of compute time. For cloud-based CI/CD platforms where compute time is billed by the minute, this difference translates to a measurable cost increase.

However, the cost comparison is not solely a function of execution time. The configuration complexity of Appium introduces a higher maintenance overhead — the test runner images for CI/CD must include not only the JDK and the Android SDK but also Node.js and the Appium server with its drivers, which increases image size and startup time. In contrast, Espresso runs entirely within the Android Gradle Plugin toolchain, requiring no additional infrastructure.

A counterpoint to these arguments is that organisations already invested in WebDriver-based test automation for web applications (e.g. using Selenium) may benefit from staffing synergies when adopting Appium for mobile testing. In such contexts, the additional configuration cost is offset by the reuse of existing competencies and tooling, which is a key factor identified in cross-platform testing studies [21, 22].

4.8. Comparison with prior empirical studies

The slowdown factor of 1.66 observed in this study is in broad agreement with prior experimental comparisons of Espresso and Appium [12, 14], which reported execution-time advantages for Espresso of similar magnitude. The novelty of the present study lies in the systematic per-module breakdown, which reveals that the slowdown factor is not uniform but varies substantially with the type of test. This finding extends previous work by providing actionable guidance: teams can estimate their projected time savings from migrating to Espresso based on the distribution of test types in their existing Appium suite.

The observation that Appium can outperform Espresso for simple visibility tests (Dashboard module) has not, to the authors' knowledge, been explicitly reported in prior work. This finding suggests that the typical narrative of Appium being uniformly slower than Espresso is oversimplified, and that the relative performance depends on the specific distribution of test types within the suite.

5. Conclusions

This paper presented an empirical comparison of the Espresso and Appium frameworks for automated UI testing of Android applications, based on the execution of an identical suite of 65 test cases in ten independent runs per framework. The main findings are summarised below.

Espresso completed the identical test suite in 437.4 s on average (stable runs), compared to 727.7 s for Appium, a slowdown factor of 1.66. Espresso's advantage is observed in five out of six functional

modules, with the largest disparity in modules requiring complex inter-screen navigation (ratios of 2.78 and 2.67 in the Profile and Medication List modules, respectively). In the Dashboard module, which contains exclusively simple visibility tests, Appium proved faster (ratio 0.71) — the only module where Appium outperformed Espresso.

In terms of run stability, Appium achieved a higher rate (80%) than Espresso (60%), although all observed failures were transient and environment-related. The standard deviation of execution times across stable runs was nearly identical for both frameworks (108.5 s and 109.1 s), indicating comparable intrinsic variability under stable operating conditions.

Implementation of the Appium test suite required substantially more configuration effort and was associated with several technical challenges related to element location, application state management, and asynchronicity handling. Espresso's declarative API yielded more concise test code with fewer implementation pitfalls.

The results suggest that Espresso should be preferred for Android-exclusive projects where the development team maintains the tests, has access to the application source code, and where execution speed is a relevant factor. Appium remains the preferred choice for cross-platform projects, for black-box testing without source code access, or in organisations with existing WebDriver-based testing infrastructure.

Limitations of this study include the use of a single hardware configuration, a single test application, and a relatively small number of repetitions (10 runs per framework). Future work could extend this comparison by including physical device testing, evaluating additional frameworks such as Maestro and Detox, analysing long-term maintenance costs in the context of evolving applications, and investigating the integration of both frameworks with mobile testing infrastructures identified in systematic mapping studies [28]. Further research could also explore the use of capture-and-replay techniques [29] and automatically generated test cases [23] as alternatives to manually scripted tests.

The research presented in this paper was conducted as part of a master's thesis at the Lublin University of Technology.

References

- [1] Mobile Operating System Market Share Worldwide, StatCounter, <https://gs.statcounter.com/os-market-share/mobile/worldwide>, [20.04.2026].
- [2] IEEE, IEEE Standard for Software and System Test Documentation, IEEE Std 829-2008, IEEE, 2008, <https://doi.org/10.1109/IEEESTD.2008.4578383>.
- [3] G. J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, 3rd ed., John Wiley & Sons, Hoboken, 2012, <https://doi.org/10.1002/9781119202486>.
- [4] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, Boston, 2010.
- [5] M. Cohn, Succeeding with Agile: Software Development Using Scrum, Addison-Wesley, Upper Saddle River, 2010.
- [6] Test Pyramid, M. Fowler, <https://martinfowler.com/bliki/TestPyramid.html>, [23.04.2026].
- [7] S. Braun, F. Elberzhager, K. Holl, Automation support for mobile app quality assurance – a tool landscape, *Procedia Computer Science* 110 (2017) 117–124, <https://doi.org/10.1016/j.procs.2017.06.129>.
- [8] Espresso — Android Developers, Google LLC, <https://developer.android.com/training/testing/espresso>, [23.04.2026].
- [9] WebDriver — W3C Recommendation, W3C, <https://www.w3.org/TR/webdriver/>, [12.03.2026].
- [10] J. Wang, J. Wu, Research on mobile application automation testing technology based on Appium, In 2019 International Conference on Virtual Reality and Intelligent Systems (ICVRIS) (2019) 247–250, <https://doi.org/10.1109/ICVRIS.2019.00068>.
- [11] G. da Silva, R. de Souza Santos, Comparing mobile testing tools using documentary analysis, In ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (2023) 1–6, <https://doi.org/10.1109/ESEM56168.2023.10304798>.
- [12] D. Zelenchuk, Android Espresso Revealed: Writing Automated UI Tests, Apress, New York, 2019.
- [13] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, Prentice Hall, 2017.
- [14] K. S. Arif, U. Ali, Mobile application testing tools and their challenges: A comparative study, In 2019 2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET) (2019) 1–6, <https://doi.org/10.1109/ICOMET.2019.8673505>.
- [15] R. K. Lenka, S. Mamgain, S. Kumar, R. K. Barik, Performance analysis of automated testing tools: JMeter and TestComplete, In 2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN) (2018) 399–407, <https://doi.org/10.1109/ICACCCN.2018.8748521>.
- [16] R. Gu, J. M. Rojas, An Empirical Study on the Adoption of Scripted GUI Testing for Android Apps, *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering Workshops* (2023), <https://doi.org/10.1109/ASEW60602.2023.00030>.
- [17] R. Coppola, M. Morisio, M. Torchiano, Maintenance of Android widget-based GUI testing: A taxonomy of test case modification causes, In 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (2018) 151–158, <https://doi.org/10.1109/ICSTW.2018.00044>.
- [18] S. K. Purushothaman, N. Kashyap, S. Agarwal, S. Iqbal, V. Behere, Unified approach towards automation of any desktop web, mobile web, Android, iOS, REST and SOAP API use cases, In 2018 International Conference on Circuits and Systems in Digital Enterprise Technology (ICCSDET) (2018) 1–5, <https://doi.org/10.1109/ICCSDET.2018.8821229>.

- [19] R. Coppola, L. Ardito, M. Torchiano, E. Alegroth, Translation from layout-based to visual Android test scripts: An empirical evaluation, *Journal of Systems and Software* 171 (2021) 110845, <https://doi.org/10.1016/j.jss.2020.110845>.
- [20] G. Lovreto, A. T. Endo, P. Nardi, V. H. S. Durelli, Automated tests for mobile games: An experience report, In 2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (2018) 73–82, <https://doi.org/10.1109/SBGAMES.2018.00015>.
- [21] D. Vajak, R. Grbić, M. Vranješ, D. Stefanović, Environment for automated functional testing of mobile applications, In 2018 International Conference on Smart Systems and Technologies (SST) (2018) 125–130, <https://doi.org/10.1109/SST.2018.8564626>.
- [22] S. Mojahed, R. Drouin, L. Sboui, ODACE: An Appium-based testing automation platform for Android mobile devices certification, In 2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (2024) 301–308, <https://doi.org/10.1109/ICSTW60967.2024.00060>.
- [23] I. Arcuschin, L. Di Meo, M. Auer, J. P. Galeotti, G. Fraser, Brewing up reliability: Espresso test generation for Android apps, In 2024 IEEE Conference on Software Testing, Verification and Validation (ICST) (2024) 185–196, <https://doi.org/10.1109/ICST60714.2024.00025>.
- [24] S. Negara, N. Esfahani, R. Buse, Practical Android test recording with Espresso Test Recorder, In 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) (2019) 193–202, <https://doi.org/10.1109/ICSE-SEIP.2019.00029>.
- [25] C. Q. Adamsen, G. Mezzetti, A. Møller, Systematic execution of Android test suites in adverse conditions, *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (2015) 83–93, <https://doi.org/10.1145/2771783.2771786>.
- [26] Q. Luo, F. Hariri, L. Eloussi, D. Marinov, An empirical analysis of flaky tests, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014) 643–653, <https://doi.org/10.1145/2635868.2635920>.
- [27] Y. D. Lin, J. F. Rojas, E. T. H. Chu, Y. C. Lai, On the accuracy, efficiency, and reusability of automated test oracles for Android devices, *IEEE Transactions on Software Engineering* 40 (2014) 957–970, <https://doi.org/10.1109/TSE.2014.2331982>.
- [28] P. Tramontana, D. Amalfitano, N. Amatucci, A. R. Fasolino, Automated functional testing of mobile applications: a systematic mapping study, *Software Quality Journal* 27(1) (2019) 149–201, <https://doi.org/10.1007/s11219-018-9418-6>.
- [29] S. Di Martino, A. R. Fasolino, L. L. L. Starace, P. Tramontana, Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing, *Software Testing, Verification and Reliability* 31(3) (2021) e1754, <https://doi.org/10.1002/stvr.1754>.
- [30] Firebase Local Emulator Suite, Google LLC, <https://firebase.google.com/docs/emulator-suite>, [15.03.2026].