Automated Software Testing Strategies for Mobile Applications: A Case Study on Appium and Airtest

Zhang Dinggen   
Wenzhou-Kean Univercity  
CSMTWenzhou, China  
1336574@wku.edu.cn

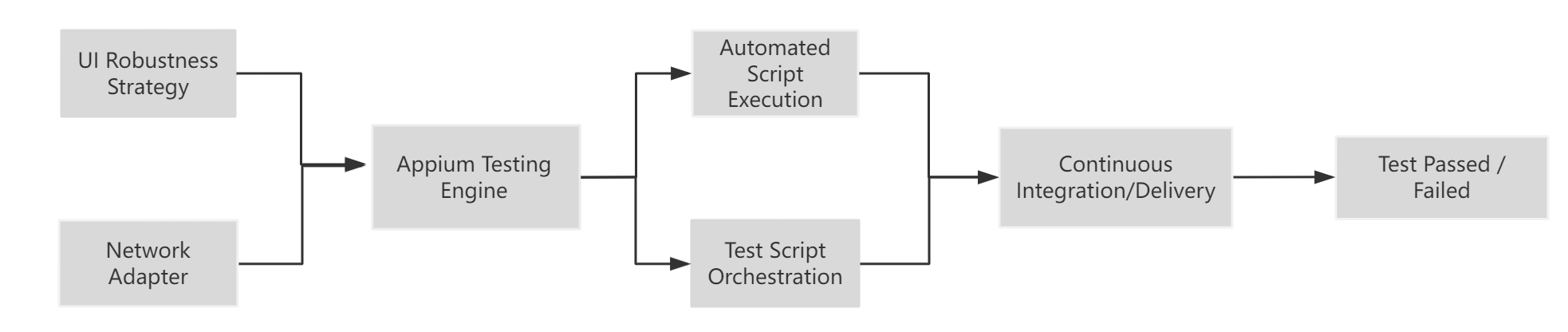
Chen Hao   
Wenzhou-Kean Univercity  
CSMTWenzhou, China  
1336564@wku.edu.cn

*Abstract*—With the rapid growth of the mobile device market and the increasing complexity and diversity of mobile applications, traditional manual testing methods are no longer sufficient to fulfill the modern need for efficient and accurate testing. This study provides a comprehensive evaluation of mobile application automation testing strategies, focusing on automated testing tools represented by Appium and comparing them with other frameworks such as Airtest. The study shows that mobile application testing faces challenges such as device heterogeneity, operating system fragmentation, unstable network conditions and cross-platform compatibility. Through experimental analysis, it is found that Appium, with its excellent cross-platform features, rich API support and good community support, has a wide range of application prospects in mobile application automation testing, which can effectively improve testing efficiency and shorten testing cycle. However, Appium may have a slight delay when dealing with new devices and OS version adaptation, and the stability of testing is greatly affected by external environmental factors. In contrast, image recognition-based frameworks such as Airtest may perform better in specific scenarios. This study suggests that development teams should introduce automated test architectures during the requirements design phase, adopt a hybrid tooling strategy, combine automated testing with continuous integration/continuous deployment processes, and strategically allocate testing resources for high-risk areas. Future research directions include framework specialization for specific technology stacks, AI-driven test generation and defect prediction, security and privacy test automation, and development of automated maturity assessment models.

Keywords—Automated Testing, Appium, Airtest, Cross-Platform, Continuous Integration, Testing Frameworks, Image Recognition, Testing Efficiency, Quality Assurance

# Introduction

With the rapid growth of the mobile device market, the complexity and diversity of mobile applications have continued to increase. Traditional manual testing approaches are no longer sufficient to meet the modern demands for efficient and precise testing. As such, the integration of automated testing strategies has become both necessary and inevitable. However, in the context of mobile application automation[1], there remain a number of unique challenges that require attention, such as: Device heterogeneity, Operating system fragmentation, Unstable and variable network conditions, Cross-platform compatibility issues. Traditional testing strategies often fall short when addressing these issues effectively and comprehensively[2][3].This study aims to conduct a comprehensive evaluation of automated testing strategies, with Appium[11] as a representative tool[4]. The focus will be on assessing their: Effectiveness, Applicability, Limitations. The ultimate goal is to enhance the quality of mobile applications, reduce testing time and cost, and provide practical guidance for testers in selecting the most appropriate automation strategies in dynamic mobile environments.

****

1. Appium-Based Automated Testing Workflow

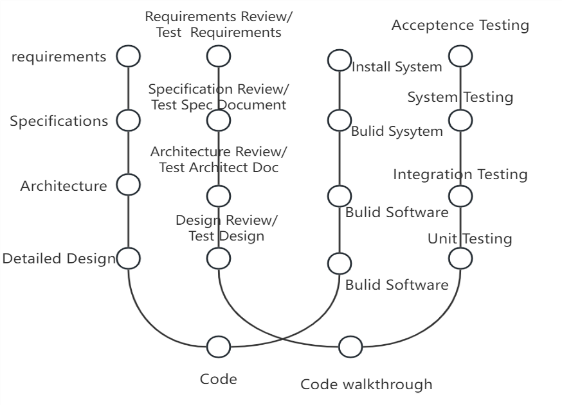
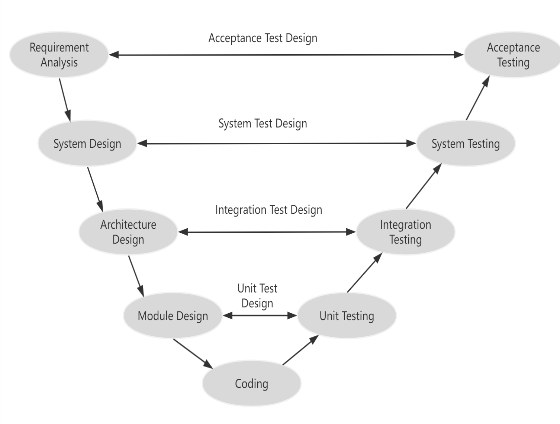
# Literature Review

## Software Testing Fundamentals

Software testing[21] is a process aimed at evaluating the correctness, completeness, and quality of software by executing it under controlled conditions[5]. Its primary goal is to identify defects and ensure that the software system meets its specified requirements[6].

## The core principles of software testing

The ability of testing to show the presence of defects, the impossibility of exhaustive testing, the cost savings of early testing, the phenomenon of defect clustering, the pesticide paradox, and the dependence of testing on the specific environment[7].The software testing life cycle consists of the phases of requirements analysis, test planning, test case[14] design, test environment setup, test execution, defect logging and retesting, and test closure[8].In the software develop life cycle (SDLC), the V model plans the development phases (user requirements analysis, system design, module design, coding implementation) in parallel with the corresponding testing phases (unit testing, integration testing, system testing, acceptance testing)[9]; while the W model is an enhanced version of the V model, emphasizing that testing activities should not be postponed until the completion of the development but should be carried out synchronously in each development phase, such as requirements validation during the requirements analysis phase and testability analysis in the requirements analysis phase, and creation of system test plans in the system design phase.



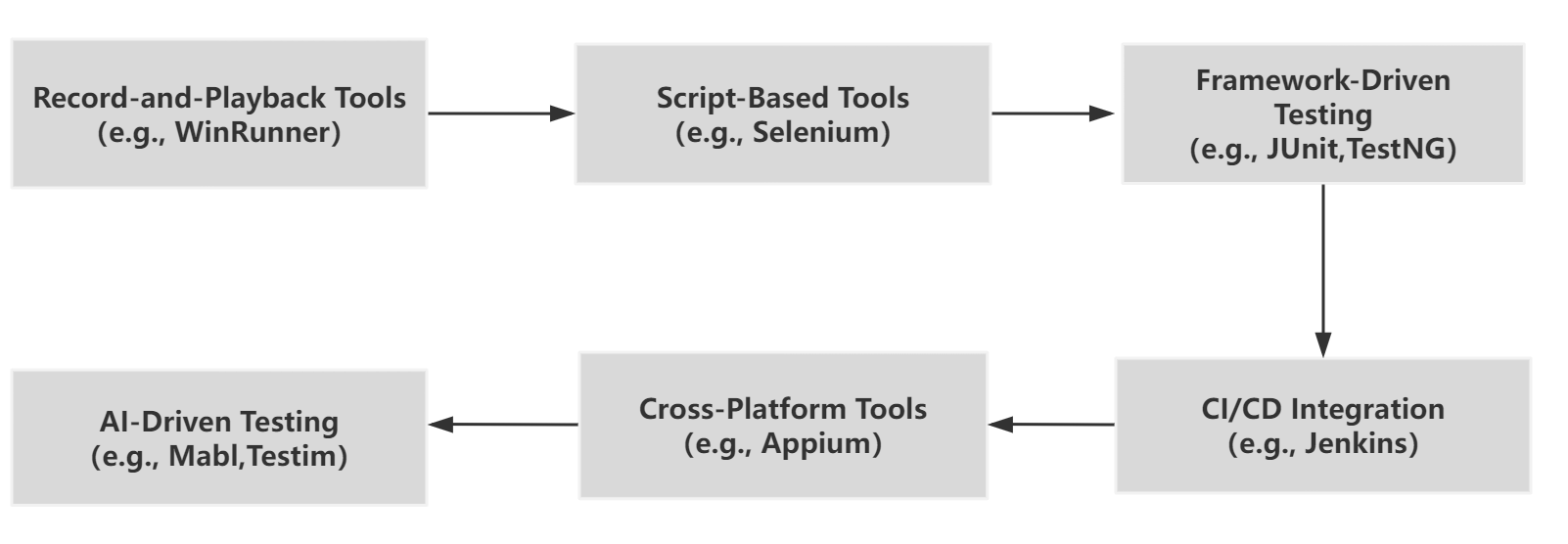
1. Comparison of V-Model and W-Model in Software Testing Lifecycle

## Mobile Applications Implications on Testing

Mobile applications can be divided into two main categories, App4Mobile are traditional desktop applications adapted to the mobile environment, such as web browsing, social networking, and collaboration tools; and Mobile Apps are context-aware applications that rely on environment-aware features to dynamically adapt to user behavior and input. Key characteristics of mobile applications and their testing implications include: connectivity, which needs to be tested under various network conditions to ensure functional stability; limited resources, which requires monitoring memory, battery and CPU usage to prevent performance degradation; autonomy, which requires evaluating the energy consumption and operational robustness of the application in low-power scenarios; and context-awareness, which needs to deal with challenges such as input explosion, and contextual modeling. and simulation testing[10].

## Evolution of Automated Testing Strategies

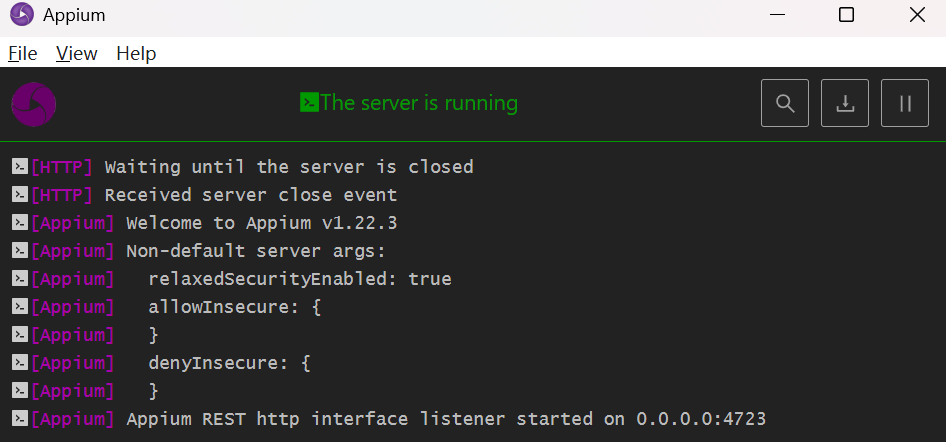
The evolution of automated testing[15] has undergone significant changes. In the early days, they relied heavily on recording and playback tools such as Mercury WinRunner and IBM Rational Functional Tester, which allowed testers to record UI interactions and replay them for regression testing. However, these tools were tightly coupled with specific UI elements and lacked the flexibility to make small changes to the interface that could lead to test failures, had limited cross-platform support, and were poorly integrated with modern development processes. They have limited cross-platform support and integrate poorly with modern development processes. With the rise of Agile and DevOps approaches, the need for more robust, maintainable, and scalable testing solutions has led to the development of script-based and framework-driven approaches, such as Selenium, which uses Java and Python for browser automation and introduces the concept of test automation frameworks that promote modularity, reusability, and CI/CD[20] integration. Today, automated testing has expanded to include mobile apps, APIs, performance testing, and AI-driven test generation, reflecting the shift from fragile UI-bound scripts to intelligent, context-aware automation strategies that support continuous integration, rapid feedback, and better quality assurance.

****

1. Evolution Roadmap of Automated Software Testing Strategies

## Appium in the Context of Mobile Automation Testing

Appium is a widely used open-source automation framework designed for testing native, hybrid, and mobile web applications across Android and iOS platforms. Built on the W3C WebDriver protocol, Appium supports cross-platform test execution using a single codebase, making it a flexible solution for teams working across diverse environments. Its client-server architecture allows test scripts to interact with mobile devices through a standardized interface, without requiring any modifications to the application under test. This non-intrusive design is especially useful for black-box testing, where access to source code is limited or unavailable. Appium supports multiple programming languages such as Java, Python, and JavaScript, and integrates well with modern development pipelines including CI/CD tools like Jenkins or GitHub Actions. Although it lacks a built-in graphical IDE, its command-based structure and rich API support enable complex test automation workflows and integration with third-party tools. Furthermore, Appium benefits from a large and active community, which contributes to rapid updates, extensive documentation, and a growing ecosystem of plugins and extensions. These characteristics make Appium a scalable, maintainable, and versatile framework for mobile automation testing, particularly in enterprise or cross-platform development contexts.



1. Appium Action Page

# Methodology

In this study, we compare the performance and usability of two automated testing frameworks, Airtest [12] and Appium[11], in a controlled testing environment.

Airtest is a cross-platform UI automation testing framework developed by NetEase, designed especially for games and apps with graphical interfaces. It utilizes image recognition technology to locate UI elements and supports scripting in Python, making it highly suitable for tasks involving visual feedback.

To ensure a fair comparison, both frameworks were configured to interact with an Android virtual device running on a Windows platform. The connection between the frameworks and the virtual device was established via Android Debug Bridge (ADB), allowing both Airtest and Appium to issue commands and capture responses in a consistent manner. The testing task was designed as follows:

* Launch the virtual Android environment.
* Use each framework to open the Settings application and navigate to the Battery options page.
* Once the Battery option is detected, trigger a screenshot of the page and save it to local storage.
* Measure the total execution time from the initiation of the test script to the successful capture of the screenshot.

This procedure was repeated multiple times for each framework to account for variability, and the average runtime was recorded as a key metric for comparison.

# Analysis

## Comparison of app base features

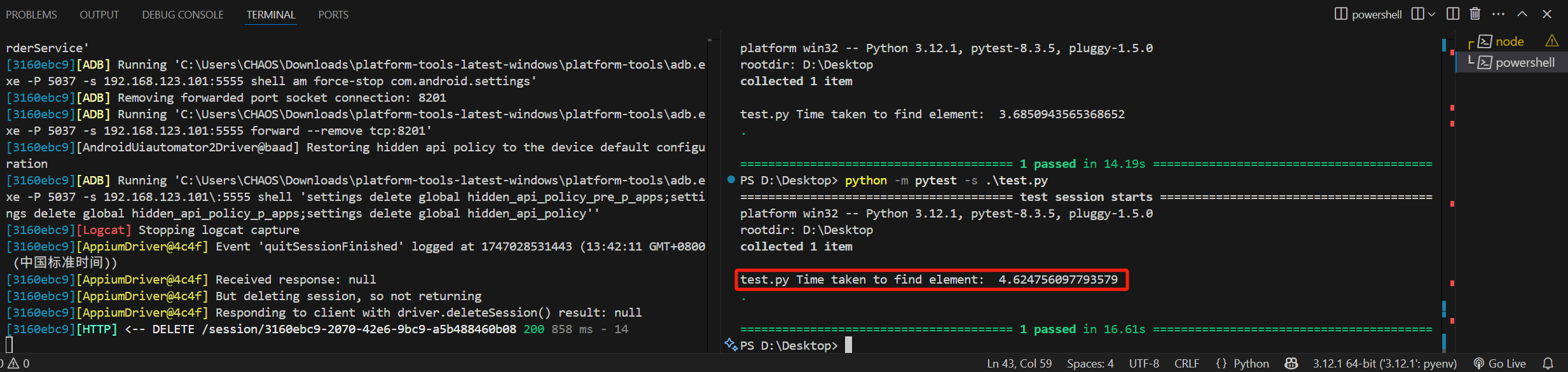
Presents a comparative analysis of the fundamental features of Appium and Airtest. Both tools support Android, iOS, and H5 platforms and have similar installation complexity. Appium holds a dominant market position, with extensive community support and abundant related resources, whereas Airtest is gaining popularity gradually but offers comparatively fewer resources. In terms of programming language flexibility, Appium supports almost all major languages, while Airtest is primarily Python-based. A notable distinction lies in development tools: Airtest provides a dedicated IDE to simplify test creation, while Appium lacks an official IDE, requiring integration with external editors. Regarding usability, Airtest demonstrates lower learning difficulty and provides essential features such as test reports, CI/CD integration, and web support—comparable to Appium. However, Appium offers additional advantages like available APIs and broader language adaptability, making it more suitable for complex, scalable automation tasks.

1. Demonstration of the characteristics of different tools

|  |  |  |
| --- | --- | --- |
| **Tool** | **Appium** | **Airtest** |
| Installation | Medium | Medium |
| Supported Platforms | Android, iOS, H5 | Android, iOS, H5 |
| Market Share | TOP1 | Gradually increasing |
| API Availability | Yes | - |
| Related Resources | Many | Few |
| Language Support | Almost all | Primarily Python |
| Development IDE | None | Yes |
| Learning Difficulty | Medium | Low |
| Test Reports | Yes | Yes |
| CI Support | Supported | Supported |
| Web Support | Supported | Supported |

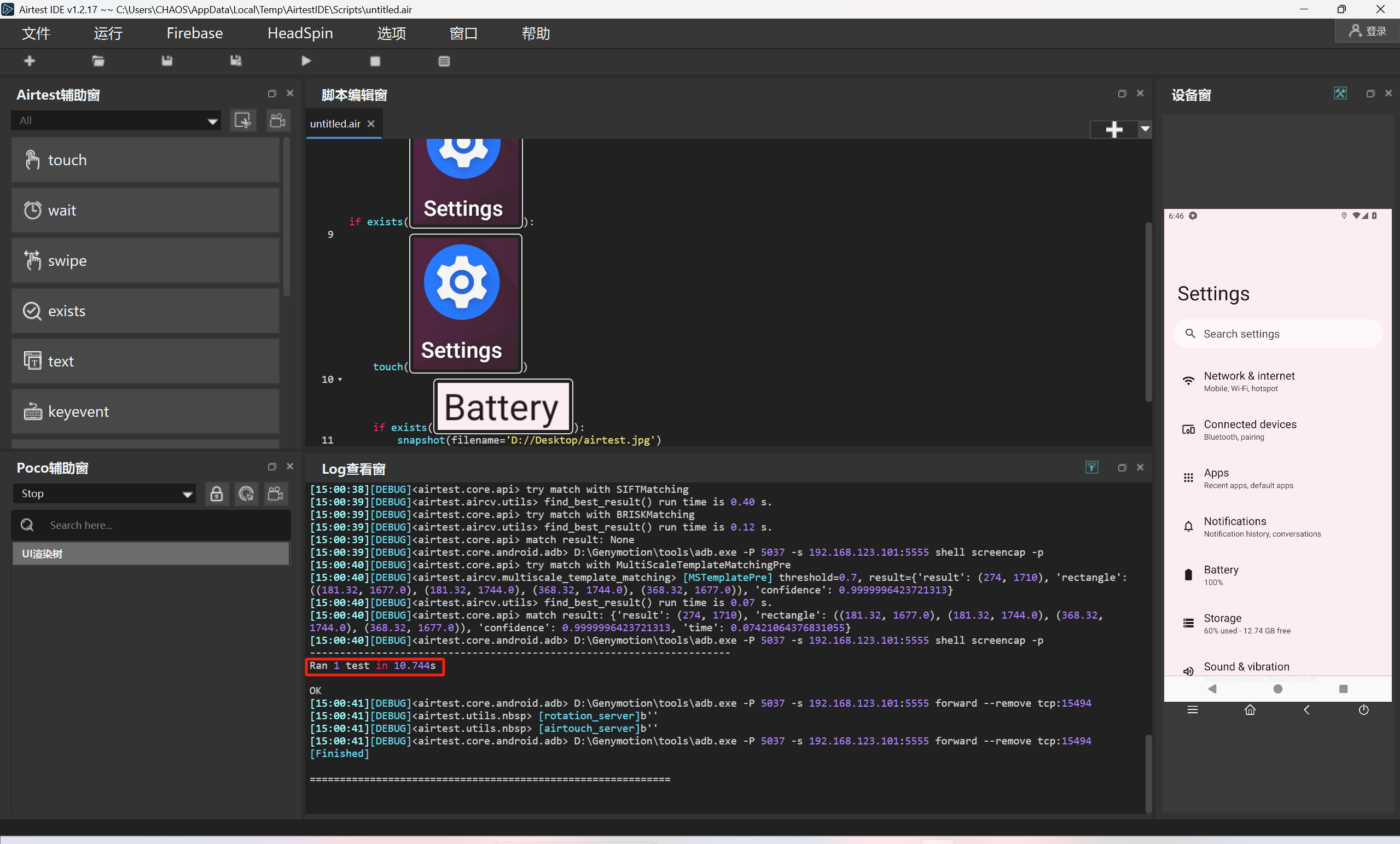
## Runtime Testing

For Appium, we need to write python scripts and we need to start the Appium server first, connect to the VM with ADB and then use python code for testing. The code needs to index the specified target tag, so it requires the tester to be familiar with the operation principle of the app under test, and requires a high threshold for use.

****

1. Appium Running Result

And Airtest also need to write code, with its code is an extension of the python code, and then after installing a specific IDE, the code can be used in the picture (screenshot in the system) to identify the app components you want to access to be tested, which greatly reduces the threshold of use, and facilitate the use of more developers.



1. Airtest Running Result

However, in terms of testing efficiency, we can find that because of the different architectural design of the two test frames, the time consumed for the same task is not the same. Appium uses element indexing to find the corresponding element in the front-end DOM tree, which is very efficient, whereas Airtest, because of its graphical programming, needs to use computer vision techniques[16] to find the corresponding tags. In summary, we believe that developers who have some familiarity with the system and need high performance, we recommend Appium, on the contrary, if you don't have a lot of basic knowledge, we recommend using the Airtest framework, which will be easier to get started in your testing process.

## Cross-platform Compatibility Assessment

|  |  |  |  |
| --- | --- | --- | --- |
| **System** | **Appium**  **Support** | **Airtest**  **Support** | **Remarks** |
| Android | Full native support | Full native support | Both frameworks handle Android well |
| iOS | With XCUITest integration | Basic support, limited | Appium supports native iOS automation |
| Web (H5) | WebDriver/Webview automation | Limited, through image-based locators | Appium offers stronger Web context handling |
| Unix  Linux | Not supported directly | Not supported | GUI apps on Unix are out of scope |

1. Testing on different Platform

Both frameworks provide full native support for Android, ensuring robust automation capabilities. For iOS, Appium demonstrates stronger support through integration with XC Test, while Airtest offers only limited functionality. In terms of web (H5) testing, Appium supports WebDriver and WebView automation, offering more reliable handling of web contexts compared to Airtest's image-based approach. Neither framework supports native automation for Unix/Linux GUI applications, which fall outside the primary scope of both tools.

## Handling of Unique Mobile Testing Challenges

In addressing complex mobile testing challenges, Appium shows several limitations that require manual intervention or third-party tools. Tasks such as network switching and battery/CPU monitoring rely on external ADB triggers or profiler tools, while context-aware testing—such as GPS simulation or handling input explosion—requires custom scripts and lacks native support. Although Appium can manage app crashes and recovery scenarios, it demands additional logic to ensure reliability. These limitations highlight the need for enhanced support mechanisms or integration with complementary tools[19] to streamline testing workflows and improve efficiency in handling dynamic mobile environments.

1. Handling of Unique Mobile Testing Challenges

|  |  |  |
| --- | --- | --- |
| **Challenge** | **Appium** | **Handling Capability** |
| Network Switching | Manual scripting needed | Moderate; external adb triggers required |
| Battery/CPU monitoring | Needs adb integration | Needs external profiler tools |
| Sensor & Context Simulation | Limited (manual GPS injection) | Only via adb scripts or custom mock |
| Input Explosion | Lacks contextual scenario management | No native support |
| App Crashes / Recovery | Partial support (need extra logic) | Retry logic easier via AirtestIDE or custom scripts |

# Discussion

## Interpretation of Results

The results indicate that while Appium’s execution speed is slightly lower than Airtest, its strengths lie in cross-platform compatibility, community support, and integration flexibility. These make it especially suitable for enterprise-level testing and CI/CD workflows.

Appium’s active open-source community ensures rapid adaptation to new Android/iOS versions and devices, enhancing test stability across fragmented environments. Its client-server architecture, though adding some latency, provides high reusability[17] and broad compatibility, making it ideal for teams working across multiple platforms.

However, this generality comes at the cost of limited support for device-specific features (e.g., sensor simulation, advanced gestures), where image-based frameworks like Airtest may perform better.

## Implications for Mobile Application Development and Testing Practices

Based on the experimental findings, Appium—despite its performance overhead—proves to be a reliable and scalable solution for integrating automated testing into the mobile development lifecycle. To maximize its value, development and QA teams should:

Plan automation early: Introduce Appium-based test architecture at the requirement/design phase, aligning it with UI components and functional modules to support early regression coverage.

Adopt a hybrid tool strategy: Combine Appium with lightweight tools like Airtest for tasks requiring image recognition, sensor simulation, or high-frequency test execution, ensuring flexibility and coverage.

Integrate with CI/CD: Use Appium in continuous integration pipelines (e.g., Jenkins, GitHub Actions) to enable automated build validation and reduce manual testing overhead.

Allocate test resources strategically: Focus automation on high-risk areas such as login, payment, and navigation flows, while using manual or exploratory testing for edge cases.

Promote collaboration: Encourage shared ownership of test scripts between developers and testers using unified coding standards and reusable Page Object models.

These practices can help reduce testing cost and risk, shorten release cycles, and ensure higher product quality across diverse mobile platforms.

## Future Research Directions

Framework specialization for specific tech stacks: Future work can focus on tailoring automation strategies to suit hybrid apps, Flutter, React Native, and game engines (e.g., Unity), which currently lack deep testing integration.

AI-driven test generation[13] and defect prediction: Leveraging machine learning to automatically generate test scripts from UI behavior or historical data, and predict high-risk failure areas, can significantly reduce manual effort and improve defect detection rates.

Security and privacy testing automation: With increasing privacy regulations and permission-sensitive app logic, research is needed on how to automate testing of permission flows, data access boundaries, and secure storage validation.

Automation maturity and evaluation models: Developing standardized frameworks for evaluating the maturity, coverage, maintainability, and ROI of automation practices will help guide test planning and tool adoption at different project scales.

Future work may focus on enhancing the usability and flexibility of Appium. Given its server-client architecture, one potential direction is to migrate its interface to a web-based environment. This could simplify the testing workflow by reducing setup complexity and enabling remote or collaborative testing, especially in cloud-based development scenarios[22]. Additionally, Appium currently lacks a native graphical interface for test creation, requiring users to write scripts manually. Introducing a visual task recording and playback tool[18] could help lower the barrier for non-programmers and streamline the automation process. These improvements would make Appium more accessible and efficient, aligning it with modern demands for low-code and user-friendly testing tools.

# Conclusion

Focusing on automated software testing strategies for mobile applications, this study analyzes the unique challenges faced by mobile application testing, such as device diversity, operating system fragmentation, changing network environments, and multi-platform compatibility issues, and evaluates automated testing strategies such as Appium.

The study finds that Appium has a broad application prospect in mobile application automation testing due to its excellent cross-platform features, rich API support and good community support. It can effectively improve testing efficiency, shorten the testing cycle, and help development teams maintain an advantage in the fierce competition in the market. However, Appium also has some limitations, for example, it may be slightly delayed when dealing with new devices and operating system version adaptation, and the stability of testing is greatly affected by external environmental factors.

Overall, automated testing has become an inevitable trend in mobile application testing, and Appium, as one of the key tools, provides testers with powerful assistance. Nevertheless, we still need to keep focusing on and improving our automation testing strategy to better adapt to the rapid development of mobile application technology and changes in the market environment, so as to provide users with personalized, high-quality and reliable mobile application experience.

##### Acknowledgment

In writing this paper, we have been fortunate to have Prof. Nasser Mustafa as our mentor in the CPS5981 SOFTWARE ASSURANCE course. With his deep expertise in software assurance, the professor provided us with a wealth of theoretical knowledge and practical advice that enabled us to gain a deeper understanding of the field. His rigorous attitude and unique insights, while helping us to master the relevant knowledge, also provided us with important guidance in the selection of dissertation topics, determination of research methodology, and dissertation writing.

##### References

1. Mamedova A N ,Bortsov S A .Automated Solution of Qualitative Image Analysis Problem Based on Laplace Variation for Mobile Application Development[J].Automatic Documentation and Mathematical Linguistics,2025,58(6):401-414.
2. Thanh D T ,Huy D T ,Su H L , et al.Mobile robot: automatic speech recognition application for automation and STEM education[J].Soft Computing,2023,27(15):10789-10805.
3. S. J S ,RS. K ,S. N .Assessment of influencing factors in selecting effective mobile applications—a drive toward automation in construction projects[J].Modeling Earth Systems and Environment,2023,9(3):3747-3754.
4. Yogashiva M .Selenium and Appium with Python:Build robust and scalable test automation frameworks using Selenium, Appium and Python (English Edition)[M].BPB Publishers:2023-05-16. DOI:10.0000/9789355518354.
5. Ardic B ,Brandt C ,Khatami A , et al.The qualitative factor in software testing: A systematic mapping study of qualitative methods[J].The Journal of Systems &amp; Software,2025,227112447-112447.
6. Polański A ,Roman A ,Zelek J .Optimal solutions for variants of graph coverage-related problems in software test design[J].Expert Systems With Applications,2025,277127216-127216.
7. Wairagade A ,Cuthrell M K .A Systematic Review of AI-Powered Software Testing in Healthcare: Methodologies, Challenges, and Future Directions[J].Journal of Engineering Research and Reports,2025,27(4):264-277.
8. Wu H .Analysis of the Effectiveness of Artificial Intelligence Technology in Defect Prediction in Software Testing[J].Innovative Applications of AI,2025,2(1):13-16.
9. Dongmo C .A Review of Non-Functional Requirements Analysis Throughout the SDLC[J].Computers,2024,13(12):308-308.
10. Muccini H, Di Francesco A, Esposito P. Software testing of mobile applications: Challenges and future research directions[C]//2012 7th International Workshop on Automation of Software Test (AST). IEEE, 2012: 29-35.
11. Alotaibi, A. A., & Qureshi, R. J. (2017). Novel framework for automation testing of mobile applications using Appium. International Journal of Modern Education and Computer Science, 9(2), 34.
12. Yu, Shengcheng, et al. "Layout and image recognition driving cross-platform automated mobile testing." 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021.
13. Jain, A., & Sinha, M. (2022). Enhancing Mobile App Testing with AI-Driven Automation Tools. Journal of Mobile Computing, 11(3), 101–113.
14. Gerasimou, S., et al. (2018). Search-based test generation for mobile apps: A survey. Information and Software Technology, 104, 1–20.
15. Moreno, L., et al. (2016). A systematic literature review of software test automation. Information and Software Technology, 76, 92–108.
16. Linares-Vásquez, M., et al. (2017). Enabling visual GUI testing for mobile apps with GATOR. Proceedings of ICSE, 540–550.
17. Rauf, A., & Memon, A. M. (2015). Test reuse in automated mobile app testing. Empirical Software Engineering, 20(1), 1–28.
18. Mesbah, A., & Prasetya, I. S. W. B. (2020). A survey of automated test oracles for GUIs. ACM Computing Surveys, 53(3), 1–40.
19. Li, L., et al. (2015). DroidBot: A Lightweight UI-Guided Test Input Generator for Android. Proceedings of Mobiquitous, 2–11.
20. Alshraideh, M., et al. (2023). CI/CD pipeline integration challenges in mobile test automation. Journal of Systems and Software, 199, 111487.
21. Kuhn, D. R., et al. (2017). Introduction to Software Testing. Cambridge University Press.
22. Mao, K., et al. (2016). Sapienz: Multi-objective automated testing for Android apps. Proceedings of ISSTA, 94–105.