Enabling Scalable Mobile Test Automation on Linux: A Case Study in a Public Financial Institution

Jonnathan Riquelmo
LESSE/PPGES/UNIPAMPA
Alegrete, RS, Brazil
jonnathan.riquelmo@gmail.com

Elrison Gomes da Silva
LESSE/PPGES/UNIPAMPA
Alegrete, RS, Brazil
elrisonsilva.aluno@unipampa.edu.br

Michael Trindade da Silva
LESSE/PPGES/UNIPAMPA
Alegrete, RS, Brazil
michaelsilva.aluno@unipampa.edu.br

Elder de Macedo Rodrigues
LESSE/PPGES/UNIPAMPA
Alegrete, RS, Brazil
elderrodrigues@unipampa.edu.br

Maicon Bernardino
LESSE/PPGES/UNIPAMPA
Alegrete, RS, Brazil
bernardino@acm.org

Abstract

Test automation is critical to ensuring software quality and security in financial institutions, particularly in public financial institutions, where the reliability of transactions is paramount. Historically, mobile test execution was constrained to Windows® servers due to technical dependencies on proprietary components. This paper presents a case study on the migration to a Linux-based architecture for mobile test automation, using Selenium Grid and Appium integrated with Jenkins CI/CD pipelines. The proposed solution enhances flexibility, scalability, and operational efficiency while maintaining strict security and compliance standards required in banking environments. The study describes the architectural decisions, implementation steps, and security adaptations undertaken to enable distributed and parallel test execution. Key outcomes include a 40% reduction in execution time, a 25% increase in functional test coverage, a shorter feedback cycle, and improved scalability. The findings provide evidence that with a structured approach and the right toolchain, robust and secure test automation is achievable even under the constraints of a highly regulated environment.

Keywords

Test Automation, Mobile Testing, Selenium Grid, Appium, CI/CD, Jenkins, Financial Systems, Linux Migration, Parallel Testing, Secure Software Testing, Infrastructure-as-Code.

1 Introduction

Software quality is a critical factor in any organization, especially in financial institutions, where application reliability and security are essential for the continuous and safe operation of transactions. In the context of a public financial institution, where the demand for quality and accuracy is particularly high, functional test automation emerges as a strategic solution to ensure high-quality software delivery with greater speed. Mobile functional testing in large organizations may combine manual, semi-automated, and automated activities, depending on infrastructure, tooling, application characteristics, and organizational constraints [5, 13, 14].

In this context, implementing a fully integrated automation solution capable of handling the complexity of banking systems becomes a strategic goal. This paper reports on the experience of implementing an automated testing architecture for mobile applications in a public financial institution, using Selenium Grid and Appium, and integrating this solution into the DevOps pipeline with Jenkins.

The main motivation for this work was to overcome the limitations of the existing infrastructure, which relied on Windows® servers to execute mobile tests due to technical constraints such as the use of specific DLLs (Dynamic-Link Libraries). The new Linux-based approach enabled greater flexibility and scalability, optimizing test performance by reducing reliance on proprietary components [12].

Moreover, integration with Jenkins facilitated continuous test automation, allowing QA (Quality Assurance) teams to develop and execute tests more efficiently and effectively. The implementation of this solution involved several stages, from the initial Proof of Concept (PoC), which validated the technical feasibility, to the adaptation of the existing architecture to meet the new requirements.

This study details each of these stages, discussing design decisions, technical challenges encountered, and the solutions adopted, as well as presenting an analysis of the results obtained with the new test architecture.

The main contribution of this paper is an empirically grounded case study of the migration of a mobile test automation infrastructure from a Windows-dependent environment to a Linux-based distributed architecture in a regulated public financial institution. More specifically, the paper contributes: (i) a description of the architectural decisions required to integrate Selenium Grid, Appium, and Jenkins under security constraints; (ii) anonymized before-and-after evidence on execution time, test coverage, feedback cycle, and execution stability; and (iii) lessons learned regarding flaky test mitigation, infrastructure scalability, and legacy-aware modernization.

To strengthen the empirical contribution of this experience report, this paper is guided by the research questions (RQs) listed in Figure 1. RQ1 examines the architectural and operational knowledge that may be transferable to similar regulated environments, whereas RQ2 assesses the effects of the migration through before-and-after operational indicators. By addressing these RQs, the paper aims to move beyond a purely technical description and provide transferable lessons for organizations facing similar constraints.

Methodologically, this work is structured as an industrial case study in software engineering, following the guidelines proposed by Runeson and Höst [15], which emphasize the importance of clearly defining the case, its context, unit of analysis, data sources, and research objectives when reporting empirical investigations

RQ1. Which architectural, security, and process decisions enabled the migration from a Windows-dependent mobile testing infrastructure to a Linux-based distributed architecture in a regulated financial environment?
RQ2. How did the migration affect test execution time, feedback cycle, functional test coverage, and execution stability?

Figure 1: Research questions of the study

in real-world settings. In this study, the case is the migration of a mobile test automation infrastructure from a Windows-dependent environment to a Linux-based distributed architecture within a regulated public financial institution. The unit of analysis is the test automation infrastructure itself, including its architectural components, CI/CD integration, operational constraints, and observed effects on the software testing process. This methodological framing is appropriate because the study investigates a contemporary phenomenon in its real industrial context, where the boundaries between technical decisions, organizational constraints, and infrastructure outcomes cannot be fully isolated.

The remainder of this paper is organized as follows. Section 2 presents the theoretical background. Section 3 describes the case study context and design. Section 4 reports the design decisions, and Section 5 presents the resulting architecture. Section 6 reports the results and answers the research questions, while Section 7 discusses infrastructure challenges. Section 8 presents the lessons learned, Section 9 discusses threats to validity, and Section 10 positions the study against related work. Finally, Section 11 concludes the paper and outlines future work.

2 Background

This section explores Selenium Grid, Appium, Parallel and Distributed Functional Testing, as well as the challenges of integrating these technologies within a highly secure and monitored network environment.

2.1 Selenium Grid

Selenium Grid is a widely used tool for the distributed execution of automated tests. It allows tests to be run on multiple machines simultaneously using a hub-nodes architecture. The hub is responsible for managing test execution and distributing them to the nodes, which are machines configured to run tests in different environments [8]. This is especially useful in scenarios where platform and browser diversity is critical to ensuring application quality.

Recent research further reinforces the central role of Selenium Grid in scalable test automation. For instance, the study by Lin and Lee (2023) [10] presents an efficient auto-scaling cross-browser testing platform built on Selenium Grid, Kubernetes, and KEDA. The authors highlight that while cross-browser testing remains a critical and resource-intensive activity in non-functional testing, auto-scaling architectures introduce significant challenges, specifically the Health-Check, Session-Queue, and Cooldown problems. By proposing targeted solutions, their work achieved a 2.27× improvement in reliability and a 61.5% reduction in execution time, compared to conventional setups. Although our solution does not currently adopt container orchestration platforms like Kubernetes, their findings validate the importance of dynamic infrastructure scaling in high-demand environments and offer valuable insights for future improvements of the bank's testing pipeline.

In the context of the public financial institution, the choice of Selenium Grid was driven by the need to perform tests across multiple operating system versions and mobile devices. However, implementation within a high-security network posed significant challenges. The bank's intranet, designed to safeguard sensitive data, imposed various restrictions on communication and access, which complicated the setup and operation of Selenium Grid.

Communication between the hub and the nodes, for example, had to be meticulously configured to ensure secure connections and prevent any compromise to network integrity. This involved additional security measures such as authentication over HTTPS protocols, the use of VPN (Virtual Private Network) tunnels, and adjustments to firewall policies to allow safe data traffic between Selenium Grid components.

2.2 Appium

Appium is an open-source tool that enables the automation of tests for mobile applications, supporting both Android® and iOS platforms. It uses an architecture similar to that of Selenium WebDriver, which facilitates integration with Selenium Grid and enables automated test execution across a wide range of mobile devices and operating systems [17].

The relevance and adaptability of Appium in diverse mobile testing contexts is reinforced by recent research efforts. Mojahed *et al.* [11] proposed ODACE, an Appium-based platform for Android certification, which demonstrated an 89% reduction in engagement time for certification testing by automating complex telecommunications scenarios and minimizing human intervention.

Complementing this, Li and Cao (2023) [9] addressed the challenges of hybrid mobile applications—combining native and web technologies, by applying Appium for API interface testing. Their approach streamlined testing workflows, enabled concurrent execution across devices, and improved statistical reporting for enterprise-scale systems.

Additionally, Chowdhury (2020) [4] focused on the configuration of Appium for iOS environments, tackling one of the platform's notable limitations: automating tests across multiple iOS devices from a single macOS machine. Together, these studies underscore Appium's robustness, cross-platform applicability, and suitability for both Android and iOS ecosystems. They also reinforce the decision to adopt Appium in our infrastructure, given its proven flexibility in certification, hybrid testing, and large-scale mobile quality assurance.

In the bank's environment, Appium was chosen for its flexibility and compatibility with the existing infrastructure, as well as its ability to run tests on both physical devices and emulators. Nevertheless, integrating Appium within the bank's intranet also presented challenges. Ensuring that tests did not compromise data security required a detailed analysis of how Appium interacted with devices and emulators.

The configuration of Android® emulators, *e.g.*, had to be adjusted to operate within the constraints imposed by the bank's network.

This included restricting access to certain system resources and implementing strict logging and visibility controls over the communication between Appium and test devices. Additionally, running tests in a restricted corporate network required the adoption of further security practices, such as isolating test environments to prevent unauthorized access.

2.3 Parallel and Distributed Functional Testing

Parallel and distributed functional testing is supported by two complementary ideas from the literature on parallel test execution and pipeline parallelism. Delgado [6] argues that parallel testing can reduce total test time by increasing the utilization of available resources, instead of reducing test coverage or sacrificing test quality. In particular, pipelining and auto-scheduling strategies allow test executions to overlap while coordinating access to shared resources, thereby improving throughput when multiple test tasks must be executed.

More recently, Chiu *et al.* [3] characterize pipeline parallelism as a general execution pattern in which stage tasks can overlap across parallel lines whenever their dependencies are satisfied. Their work also highlights that efficient parallel execution depends not only on running tasks concurrently, but also on appropriate scheduling, resource coordination, and the selection of parallel lines or threads.

In the context of this study, these principles are reflected in the orchestration of mobile functional tests across Selenium Grid nodes and Appium-based execution environments. Implementing parallel and distributed testing was crucial for dealing with system complexity, Android version fragmentation, and the need to execute test suites across multiple mobile configurations. However, executing tests concurrently within a highly monitored banking network also introduced additional challenges. Each test execution had to be carefully managed to ensure that the generated traffic did not violate institutional security policies. This required secure communication channels between the Selenium Grid hub and its nodes, controlled access to test environments, monitoring of network resource usage, and adjustments to the institution's network architecture. Practices such as network segmentation and the use of internal VPNs, specifically HPE Aruba Networking in this case, were therefore necessary to support simultaneous test execution without compromising security, performance, or compliance requirements.

2.4 Flaky Tests in Mobile Testing Environments

In test automation, a *flaky* test is defined by its non-deterministic behavior, *i.e.*, it may pass or fail across different executions despite no changes in code, environment, or input data. Parry *et al.* [13] highlight that flaky tests undermine pipeline reliability and hinder systematic characterization. This type of instability compromises the reliability of continuous integration pipelines, makes it difficult to identify genuine failures, and results in operational costs due to the need for rerunning tests and manually analyzing incorrectly reported failures [13].

Although flaky tests may occur at various levels, such as unit and integration tests, they are most frequently observed in user interface tests. Factors such as variations in element load times, network dependencies, event concurrency, and emulator instability make UI tests particularly prone to flakiness [7, 14]. In mobile testing, especially for hybrid applications, this situation worsens because of the combination of native and web layers, which increases the complexity of automation.

In the context of this research, automated tests with Appium showed a high occurrence of instability during execution on Android emulators running within a restricted corporate network. This rendered their direct integration into the institution's main CI/CD pipeline unfeasible, as the pipelines follow a blocking model where any failure immediately interrupts the continuous delivery process. As a mitigation strategy, interface tests were isolated in parallel jobs.

For each Jenkins GUI test job, teams defined a promotion threshold, typically between 70% and 85%, calculated as the proportion of completed GUI test cases that passed within that job. An artifact could proceed to the main pipeline only when its job reached the corresponding team-specific threshold. There, the code goes through the final stages of building, signing, and distribution. This approach significantly reduced the impact of flaky tests on delivery pipeline stability and improved the reliability of the automated testing process in a regulated environment.

The threshold-based strategy should be interpreted as a pragmatic containment mechanism rather than a definitive solution for flaky tests. To avoid masking real defects, failing executions are not automatically ignored; instead, they are classified according to failure symptoms such as emulator instability, timeout, network unavailability, synchronization issues, and application-level defects. Tests repeatedly failing due to infrastructure-related causes are quarantined for maintenance, while persistent application-level failures block promotion to the main pipeline. This distinction is important because flaky tests may reduce confidence in CI/CD results and increase manual triage effort if not systematically monitored.

3 Case Study

This study follows the principles outlined by Runeson and Höst [15] and Wohlin [18], who emphasize the need for case studies in software engineering to be clearly defined in terms of their unit of analysis, context, and objectives. In this work, the case under investigation is the process of migrating and evolving the mobile test automation infrastructure in a public financial institution.

3.1 Case and Organizational Context

The public financial institution studied in this paper, whose name is omitted due to a Non-Disclosure Agreement (NDA), is a state-owned Brazilian financial institution with more than 90 years of operation. It is one of the largest regional banks in the country, with a broad presence across the southern region of Brazil and a strong emphasis on digital transformation in recent years.

The institution's Technology, Innovation and Digital Transformation Directorate is responsible for managing the development, testing, deployment, and monitoring of a wide range of critical systems, including core banking, mobile applications, enterprise resource platforms, and customer service technologies. The directorate supports both mainframe and distributed architectures and has adopted DevOps practices in recent years to modernize its

software engineering processes. The mobile banking team plays a pivotal role in ensuring quality assurance in Android and iOS applications, which collectively serve millions of users.

The Institution's Information Technology (IT) has an organizational structure that includes the Technical Support Management within the Systems Development Unit belonging to the Technology, Innovation and Digital Transformation Directorate. These areas are responsible for various activities related to IT within the bank.

Within this organizational structure, the Technical Support Management provides support for the institution's IT systems and infrastructure, while the Systems Development Unit develops and maintains software solutions. Together, these units support the operation, availability, and continuous improvement of the bank's digital services.

3.2 Case Study Design

The case study reported here is descriptive and exploratory in nature. It aims to describe the process and outcomes of a strategic shift in testing infrastructure, focusing on: (1) Overcoming platform dependency on Windows®; (2) Enabling distributed and parallel execution on Linux nodes; (3) Integrating automation into CI/CD pipelines using Jenkins; (4) Preserving high levels of security, isolation, and auditability.

The unit of analysis is the automation infrastructure itself, while the data sources include performance indicators (*e.g.*, test execution time, coverage, feedback cycles), architecture diagrams, and internal documentation from the mobile quality assurance team. Due to the institution's confidentiality policies, quantitative metrics are only discussed in aggregated or illustrative terms.

The study comprised six stages: (1) characterization of the legacy infrastructure; (2) identification of migration constraints; (3) implementation of the PoC; (4) deployment of the Linux-based architecture; (5) collection and comparison of before-and-after indicators; and (6) synthesis of architectural decisions and lessons learned.

The study was conducted as a single-case embedded case study, in which the main case is the migration of the mobile test automation infrastructure and the embedded units are three representative mobile applications maintained by different teams. The data collection covered executions performed before and after the migration, using Jenkins build logs, Selenium Grid execution metadata, internal QA reports, and architecture documentation. To preserve confidentiality, application names, repositories, internal services, and exact infrastructure identifiers were anonymized. However, the quantitative indicators were preserved in relative and aggregated form, allowing comparison between the legacy and migrated environments. The analysis followed a before-and-after design, comparing the available indicators for execution time and test coverage across the selected applications, complemented by operational evidence regarding feedback cycle, success rate, scalability, queue behavior, and flaky test mitigation.

Execution success rate was defined as the proportion of completed Selenium Grid sessions that finished successfully during the observation period. This session-level metric differs from the team-specific promotion threshold, which was calculated from passed GUI test cases within each Jenkins job.

4 Design Decisions

The design and implementation of the new automated testing infrastructure involved a set of strategic decisions. Each addressed specific technical, operational, or security challenges present in the previous environment.

A typical execution scenario starts when a developer submits changes to the internal repository and triggers a Jenkins job associated with a mobile application. Jenkins checks out the corresponding project, selects the target environment and Android Virtual Device profile, and sends the execution request to the Selenium Grid Hub. The hub then allocates the test session to an available Linux node running Appium and the required Android emulator version. After the test suite is executed, logs, screenshots, execution status, and timing metadata are collected and returned to Jenkins, where reports become available to QA analysts and developers. This workflow replaced the previous model in which test execution was tied to a limited set of Windows servers and proprietary authentication components.

The key design decisions are detailed below:

DD1. Migration from Windows® to Linux-Based Test Execution – *Rationale:* The existing infrastructure relied on proprietary DLLs developed in C#, limiting test execution to Windows® environments. These dependencies were incompatible with Linux and impeded scalability and automation. Migrating to Linux allowed for the elimination of these constraints, enabled the use of open-source tools, and reduced reliance on proprietary software;

DD2. Reconfiguration of Jenkins for CI/CD Integration – *Rationale:* Jenkins jobs were adapted to integrate seamlessly with the new Linux-based infrastructure. This included updating pipeline scripts, repository configuration (SVN/ GitLab), environment selection, and Gradle task execution (*e.g.*, `buildFunctionalTest`) to support distributed execution triggered by Jenkins;

DD3. Proof of Concept (PoC) to Validate Feasibility – *Rationale:* Before full migration, a PoC (Figure 2) was conducted using Selenium Grid and Appium on Linux servers. The goal was to verify the technical viability of executing Android® emulators, running functional test suites, and integrating with Jenkins in the target environment. The PoC confirmed the feasibility of the approach and revealed performance and resource management benefits;

DD4. Replacement of DLLs with Secure API-based Authentication – *Rationale:* Since the original authentication mechanisms depended on Windows-only DLLs, new APIs were developed to perform the same functions using HTTP(S), compatible with Linux. These APIs ensured secure access to internal systems while maintaining compliance with the bank's security policies;

DD5. Refactoring of Internal Testing Frameworks – *Rationale:* To accommodate the new execution environment, parts of the internally developed test automation frameworks (for web and mobile) were refactored. The updated code-base supports cross-platform execution and enhances maintainability while preserving legacy test logic and patterns.

Table 1: Summary of the main architectural decisions and their adaptation rationale

Decision	Constraint Addressed	Adopted Mechanism	Alternative Considered	Operational Consequence
Replace DLL-based authentication	Dependency on Windows-specific components	Linux-compatible HTTP(S) authentication APIs	Windows DLLs or a platform-specific compatibility layer	Cross-platform execution and reduced dependency on proprietary components
Use headless Android emulators	Limited CPU and memory capacity	Android emulators running without a graphical interface on dedicated Linux nodes	Graphical emulator execution	Lower resource overhead and greater emulator capacity per node
Separate GUI test jobs	Flaky tests disrupting blocking CI/CD pipelines	Parallel GUI test jobs with team-specific promotion thresholds	Direct inclusion of GUI tests in the main blocking pipeline	Reduced pipeline disruption and independent handling of unstable tests
Centralize test orchestration	Limited scalability and static allocation of test executions	Selenium Grid Hub with dynamic allocation to available execution nodes	Fixed allocation of tests to specific servers or environments	Improved resource utilization and dynamic session distribution
Integrate distributed execution with Jenkins	Existing CI/CD jobs were designed for the Windows-based environment	Updated Jenkins jobs, pipeline scripts, repository settings, and Gradle tasks	Manual execution or Windows-specific Jenkins jobs	Automated triggering and reporting across distributed Linux nodes
Refactor internal testing frameworks	Platform-specific implementation and legacy dependencies	Cross-platform refactoring while preserving existing test logic	Complete framework replacement or continued Windows dependency	Improved portability and maintainability with lower migration risk
Validate the migration through a PoC	Uncertainty about emulator execution and Jenkins integration on Linux	Small-scale Selenium Grid and Appium environment	Immediate full-scale migration	Reduced implementation risk and early identification of resource constraints

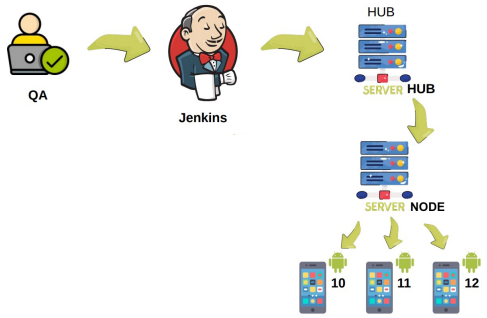

Figure 2: High-level flow of the proof of concept

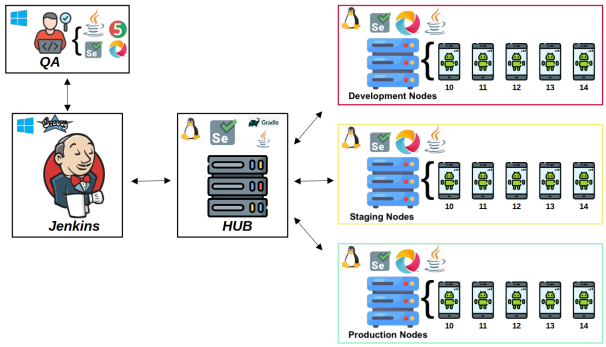

Figure 3: Current software architecture

Table 1 summarizes the main architectural decisions, the constraints they addressed, the adopted mechanisms, and their operational consequences. This decision-oriented representation is intended to help practitioners assess how the architecture could be adapted to environments with similar legacy, scalability, and security constraints.

5 Software Architecture

Figure 3 presents the architecture of the new solution, designed to maximize the efficiency, scalability, and security of automated testing within a critical banking environment. At the core of the architecture is the Selenium Grid Hub, configured on a Linux server, which acts as the central orchestration point for test execution. Test projects are developed by QA teams and versioned in the bank's internal SVN or GitLab repositories.

In the Jenkins jobs, the corresponding repository is specified for cloning, along with the target environment (development, staging, or production) and the AVD (Android Virtual Device). The Hub receives the test requests triggered by Jenkins, and after cloning the project, it essentially runs a Gradle command such as buildFunctionalTest, distributing the test tasks to available nodes—servers configured with Android® emulators running different OS versions.

Each node is responsible for emulating specific Android® environments, ranging from older to the most recent versions of the operating system. This ensures comprehensive test coverage, validating the applications across a wide variety of configurations. The nodes also operate in headless mode, *i.e.* without a graphical interface, which improves processing efficiency by reducing resource consumption [16]. The headless mode is particularly important to optimize memory and CPU usage on the servers, increasing the ability to execute multiple tests concurrently.

The test distribution performed by Selenium Grid is one of the most significant efficiency gains of this architecture. The intelligent orchestration handled by the hub allows test executions to be redirected to idle nodes, even when those nodes belong to different environments. This occurs transparently, since the Android® emulators are identical, and the separation of servers into different labels is merely semantic—used to organize the infrastructure and generate grouped test reports based on the type of APK (Android Application Package) being tested, whether it belongs to the development, staging, or production version. Across the three analyzed applications, the elapsed time required to execute the selected test suites decreased by an average of 40% after the migration.

Another important aspect of the architecture is the continuous integration with Jenkins, which acts as the coordinator of test jobs. Jenkins connects directly to the Selenium Grid hub, triggering the execution of each test task and generating final reports

Table 2: Evolution of the infrastructure used for mobile test execution

Scenario	Server Type	OS	CPU	RAM	Storage	Execution Capacity	Architectural Role
Legacy Infrastructure (Before Migration)	1× Execution Node	Windows Server 2019	1× 12 vCPUs	1× 64 GB	1× 500 GB	Limited capacity for the parallel execution of Android emulators.	Mobile test execution depended on Windows-specific components, including proprietary DLLs for internal authentication. These components were subsequently replaced with Linux-compatible HTTP(S) APIs. The infrastructure was difficult to scale.
	1× Central Server	Windows Server 2019	1× 8 vCPUs	1× 16 GB	1× 500 GB	Centralized test orchestration and Jenkins integration. The server was not dedicated exclusively to mobile testing, as it also executed web tests and built jobs for the institution's C# and Java projects.	Responsible for coordinating test executions and consolidating their results. The maximum feedback cycle reached approximately **18 working hours**, corresponding to three working days at the institution.
Proof of Concept	1× Emulation Node	Oracle Linux 9	1× 6 vCPUs	1× 12 GB	1× 120 GB SSD	Execution of two Android emulators simultaneously.	Environment used to validate the technical feasibility of distributed mobile test execution using Selenium Grid and Appium on Linux. The environment was intentionally undersized because it was a PoC conducted with limited computational resources.
	1× Orchestration Hub	Oracle Linux 9	1× 4 vCPUs	1× 8 GB	1× 100 GB SSD	Orchestration of test executions initiated by Jenkins.	Hosted the Selenium Grid, Jenkins, and the new API-based authentication mechanisms on Linux.
Final Infrastructure	3× Emulation Nodes (Development / Staging / Production)	Oracle Linux 9	3× 12 vCPUs	3× 48 GB	3× 500 GB SSD	Execution of up to five Android emulators simultaneously per node, distributed across the development, staging, and production environments. Each server is dedicated exclusively to Android emulator execution.	Distributed execution in *headless* mode, enabling more efficient CPU and memory utilization, greater parallelism, and dynamic session distribution through the Selenium Grid Hub.
	1× Orchestration Hub	Oracle Linux 9	1× 4 vCPUs	1× 16 GB	1× 250 GB SSD	Centralized management of distributed test executions.	Responsible for Jenkins integration and intelligent session distribution across the available nodes. The final infrastructure contributed to an approximate **40% reduction** in execution time and reduced the maximum feedback cycle to **six hours**.

upon completion. Communication between Jenkins and the hub is streamlined and direct, minimizing failures and delays even when running large test suites. Once the tests are completed, the results are aggregated at the hub, processed, and sent back to Jenkins, where they are made available for QA and development teams to analyze.

Furthermore, the architecture was designed with strict concern for data security and integrity. In a banking environment, security is a top priority, and the adopted architecture reflects this. All communication between the hub and the nodes, as well as between Jenkins and the hub, is secured through authentication mechanisms (internal ticket-generation systems for tests, logging, and traceability for audit purposes) and SSL (Secure Sockets Layer) encryption, ensuring that sensitive data is not exposed during test execution. Each node is isolated to prevent unauthorized access, and encryption of data in transit ensures that confidential information remains protected.

Finally, the modular design of the architecture facilitates scalability. New nodes can be easily added to the existing structure, allowing test execution capacity to be increased as needed, without requiring complex reconfiguration. This flexibility enables the bank to adjust its infrastructure according to demand, scaling up by adding servers during periods of high test volume or scaling down by deactivating nodes when usage is low.

Table 2 summarizes the evolution of the mobile test automation infrastructure, comparing the server configurations, execution capacity, and architectural roles of the legacy Windows-based environment, the PoC, and the final Linux-based architecture.

6 Results and Discussion

The implementation of the new automated testing architecture brought marked improvements compared to the legacy infrastructure. Before the migration, mobile tests were executed exclusively on Windows® servers, which introduced critical limitations, including strong dependencies on proprietary components, elevated test execution times, and poor scalability within Jenkins pipelines. The Linux-based distributed architecture addressed these restrictions, offering higher throughput and operational efficiency.

Additionally, test monitoring was previously limited to text-based Jenkins console logs, making real-time analysis difficult. With Selenium Grid, the QA team gained a centralized, visual dashboard of all active nodes and emulators, as illustrated in Figure 4.

To better substantiate the reported improvements and address empirical verification, we established a quantitative baseline. Due to Non-Disclosure Agreements (NDA) and strict banking regulations, the specific corporate identifiers of the systems have been anonymized. Table 3 presents the comparative baseline of performance metrics across three representative software products of the institution, each containing a comprehensive suite averaging 300 functional test cases. The values were anonymized and rounded to preserve confidentiality while preserving the magnitude and direction of the observed changes across the selected applications.

As observed in Table 3, the total test execution time dropped by approximately 40% across all analyzed applications. This performance improvement is associated with the combined effect of distributed execution, headless emulator operation, and more efficient orchestration of available nodes by Selenium Grid. Furthermore, the Selenium Grid Hub actively orchestrated test distribution to idle

Table 3: Anonymized and aggregated performance and coverage indicators before and after migration

Anonymized App	Execution Time (min)		Reduction	Test Coverage		Relative Gain
	Legacy (Windows)	New (Linux)	(%)	Legacy (Windows)	New (Linux)	(%)
App A (Core Banking)	120.0	72.0	40.0%	50.0%	62.5%	+25.0%
App B (Payment System)	45.0	27.0	40.0%	40.0%	50.0%	+25.0%
App C (Investments)	30.0	18.0	40.0%	60.0%	75.0%	+25.0%
Overall Average	**65.0**	**39.0**	**40.0%**	**50.0%**	**62.5%**	**+25.0%**

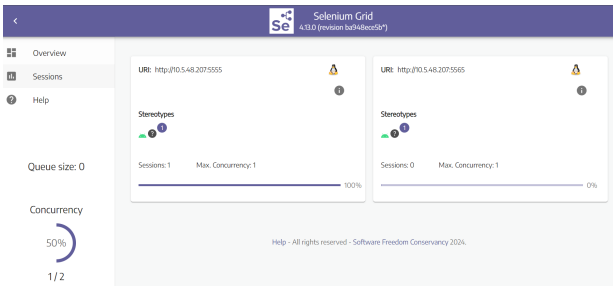

Figure 4: Overview provided by the Selenium Grid Hub

emulators across separate hardware nodes, decoupling execution from the semantic barriers of development, staging, or production pipelines. Figure 5 shows an active test session being monitored during the PoC.

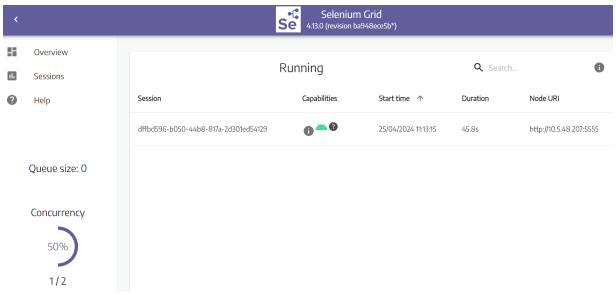

Figure 5: Test being executed and visualized via the hub

Key Performance Indicators (KPIs) also revealed a 25% increase in functional test coverage. The analyzed projects contained test suites with an average of 300 functional test cases, approximately 150 of which were processed by each node during distributed execution. Each node supported up to five concurrent Android emulator sessions; therefore, the 150-test figure represents the workload processed per node rather than the number of simultaneously executing test cases.

Across the observation period, 98% of the Selenium Grid test sessions completed successfully. This aggregate session-level indicator is distinct from the team-specific promotion thresholds of 70% to 85%, which were calculated from test-case outcomes within individual Jenkins jobs. Reaching a promotion threshold did not automatically classify the remaining failures as acceptable: infrastructure-related and flaky failures were investigated separately, while persistent application-level failures remained blocking. This level of stability contributed to shortening the engineering feedback cycle, defined as the time window required to resolve code defects after an automated report. Under the Windows setup, the feedback cycle upper limit frequently stretched up to 18 hours. With the parallel Linux execution, this limit was compressed to a maximum of 6 hours (a 66.7% reduction), effectively fitting within a standard single-day working shift.

Continuous integration within Jenkins was similarly enhanced. Tests are now triggered dynamically as an integrated step of the CI/CD pipeline, guaranteeing that code changes are verified prior to branch merging. This automation mitigates deployment risks, which is vital for large-scale financial applications. Figure 6 highlights the metadata of a successfully initiated functional testing session embedded in the development pipeline.

Figure 6: Session information displayed on the hub

Consistent with recommendations on the careful use of statistical analysis in software engineering studies [2], the evaluation emphasizes aggregated before-and-after indicators rather than controlled experimentation or full statistical inference. For each anonymized application, we report the available comparative indicators related to execution time and test coverage, complemented by operational evidence regarding feedback cycle, execution success rate, and scalability. This analysis does not aim to establish causal generalization beyond the studied organization, but to provide transparent evidence of the observed operational gains in a real-world regulated environment.

Finally, architectural scalability was achieved. Instead of relying on a fixed, high-maintenance cluster of Windows® servers, the new framework allows system administrators to scale out by attaching additional compatible Linux nodes to the grid. This elastic capability ensures stable execution queues even during high-demand regression periods, such as pre-release code freezes.

While the proprietary nature of the financial network restricts the public release of full raw logs and internal code repositories, the aggregated empirical metrics presented in Table 3 provide evidence of substantial stability and velocity gains in the migrated ecosystem.

6.1 Answering the Research Questions

The migration was motivated by platform dependency, limited scalability, and operational bottlenecks in the legacy Windows-based infrastructure. Proprietary DLLs constrained mobile test execution to Windows servers, while the institution required a solution compatible with its security, auditability, and compliance requirements. This context informed the architectural decisions examined in RQ1 and the operational outcomes assessed in RQ2.

RQ1. Enabling mobile test automation in a regulated financial environment required a combination of architectural, security, and process-oriented decisions. Selenium Grid was adopted as the orchestration layer, while Appium nodes with Android emulators were deployed on Linux servers and integrated with Jenkins. Windows-dependent DLL authentication was replaced with secure HTTP(S) APIs, and internal testing frameworks were refactored to support cross-platform execution. The architecture also incorporated encrypted communication, access control, VPN-based connectivity, node isolation, logging, and traceability mechanisms to meet institutional security requirements. To mitigate flaky GUI tests, these tests were isolated in parallel jobs, and artifacts were promoted to the main pipeline only after reaching predefined success thresholds. Together, these decisions enabled a more scalable and maintainable infrastructure without compromising the security and operational controls required by the institution.

RQ2. The migration positively affected the operational indicators analyzed in the case study. The anonymized before-and-after comparison showed an average 40% reduction in test execution time across the selected mobile applications, while functional test coverage increased by an average of 25% relative to the legacy baseline, corresponding to an average increase of 12.5 percentage points. The upper limit of the feedback cycle for defect analysis and resolution decreased from 18 hours to approximately 6 hours, corresponding to a 66.7% reduction. The aggregate execution success rate reported in Section 6 was consistent with greater stability and predictability in mobile test execution. This indicator should be distinguished from the team-specific thresholds used to decide whether artifacts could proceed to the main pipeline. Overall, the results show that the migration enabled faster feedback, broader automated coverage, and more efficient use of testing resources.

On the whole, the answers to the two RQs show that the migration required a security-aware combination of architectural and process decisions and resulted in improvements across the main operational indicators. Although the findings are bounded by the single-case design and confidentiality restrictions, they provide practical evidence that Linux-based distributed mobile test automation can be a viable modernization path for organizations facing similar legacy, scalability, and regulatory constraints.

7 Infrastructure Considerations and Challenges

Implementing a distributed and parallel automated testing architecture within a highly secure banking network is a complex task that requires overcoming various technical and infrastructure challenges. The migration of mobile tests to Linux servers, *e.g.*, required not only deep knowledge of the technologies involved but also careful adaptation of security policies and authentication processes.

Configuring Android® emulators as Selenium Grid nodes, for instance, demanded close attention to the various constraints imposed by the bank's network security. This included implementing strict access controls, encrypting data in transit, and adapting emulators to operate in isolation, ensuring that each test environment remained isolated and protected against unauthorized access.

The integration of Selenium Grid and Appium with Jenkins, which runs on a Windows® environment, was another critical aspect. Establishing communication between these different operating systems within a highly controlled network posed significant challenges. It was necessary to develop solutions that would allow secure and efficient communication between the Linux servers hosting the Selenium Grid and the Windows® server running Jenkins, ensuring that test automation functioned smoothly and cohesively.

In summary, implementing automated testing in a banking environment goes beyond the technical configuration of tools. It also demands careful adaptation to stringent security requirements and thoughtful integration with the existing infrastructure, all while ensuring that tests can be executed efficiently and securely, without compromising data integrity or network security.

8 Lessons Learned

Beyond the technical outcomes of the migration, this case study provides practical lessons for organizations that need to modernize mobile test automation under legacy, scalability, and regulatory constraints. These lessons are derived from the architectural decisions, operational challenges, and empirical observations reported.

LL1 – Legacy dependencies should be isolated before infrastructure migration. The migration from Windows® to Linux was not only an operating system change, but also a legacy disentanglement effort. The previous infrastructure depended on Windows-specific DLLs for internal authentication and communication mechanisms, which constrained test execution to a limited set of servers. Replacing these dependencies with secure API-based mechanisms was a necessary step to enable cross-platform execution. Therefore, organizations planning similar migrations should first identify platform-specific dependencies and isolate them behind portable interfaces before moving test execution to a new infrastructure.

LL2 – GUI tests should not be blindly embedded into blocking CI/CD stages. Mobile GUI tests, especially those executed on emulators or hybrid applications, tend to be more unstable than lower-level tests because they depend on GUI states, synchronization events, emulator behavior, and network conditions. In the studied environment, directly placing these tests inside the main blocking CI/CD pipeline would increase the risk of interrupting the delivery process due to nondeterministic failures. A more sustainable strategy was to execute GUI tests in separate parallel jobs and promote artifacts to the main pipeline only when predefined success thresholds were reached. This approach helped balance delivery confidence, infrastructure cost, and pipeline stability.

LL3 – Security constraints must be treated as architectural drivers, not as post-deployment adaptations. In a regulated financial environment, security requirements directly shape the test automation architecture. Network segmentation, VPN-based connectivity, encrypted communication, access control, logging, traceability, and node isolation were not optional additions; they

were core architectural requirements. Treating security as an architectural driver from the beginning avoided later rework and ensured that distributed test execution could operate within the institution's compliance constraints.

LL4 – Observability is essential for scaling distributed mobile test execution. As test execution becomes distributed across multiple nodes, emulators, and environments, text-based logs are no longer sufficient for effective monitoring and troubleshooting. The Selenium Grid dashboard improved visibility over active sessions, node availability, queue behavior, and emulator usage. This observability was important not only for debugging failures, but also for supporting operational decisions about scalability, resource allocation, and test scheduling.

LL5 – Anonymized operational metrics can still support empirical transparency under NDA constraints. Although confidentiality restrictions prevented the disclosure of raw logs, internal repositories, infrastructure identifiers, and application names, the study was still able to report aggregated and anonymized before-and-after indicators. These metrics supported a transparent analysis of execution time, test coverage, feedback cycle, and execution stability without exposing sensitive institutional information. This suggests that industrial case studies in regulated environments can still provide empirical value when they clearly explain what was anonymized, which indicators were preserved, and how the evidence supports the reported findings.

Overall, these lessons indicate that the success of a mobile test automation migration depends not only on adopting tools (Selenium Grid, Appium, and Jenkins), but also on managing legacy dependencies, pipeline stability, security constraints, observability, and empirical reporting practices. In this sense, the main contribution of this study lies in showing how established test automation technologies can be combined and adapted to operate reliably in a legacy-constrained and highly regulated institutional environment.

9 Threats to Validity

Here, we discuss the main threats to validity that may affect the interpretation, reliability, and transferability of the findings [18].

Construct Validity: The study relies on operational metrics extracted from CI/CD logs and internal QA reports. To reduce measurement bias, metrics were defined before analysis and computed consistently for the legacy and migrated environments.

The team-specific promotion thresholds, which ranged from 70% to 85%, should not be interpreted as the aggregate execution success rate reported in Section 6. Because an artifact could proceed despite some unsuccessful GUI test cases, the 98% aggregate session completion rate may not fully represent application-level reliability if infrastructure-related, flaky, and product-level failures are not correctly distinguished. To mitigate this threat, unsuccessful executions were manually classified by failure type, persistent application-level failures remained blocking, and tests repeatedly affected by infrastructure instability were quarantined for maintenance. Nevertheless, variation in thresholds across teams limits direct comparability and should be considered when interpreting the results.

Internal Validity: Since this is a before-and-after case study, improvements cannot be attributed exclusively to the operating system migration. Other factors, such as test suite refactoring, Jenkins job reconfiguration, and emulator optimization, may also have contributed to the observed gains.

External Validity: The study was conducted in a single public financial institution with strict security and compliance constraints. Therefore, the results may not generalize directly to organizations with different regulatory, architectural, or infrastructure contexts.

Reliability: Although source code, raw logs, and internal configuration files cannot be released, the paper provides an anonymized architecture diagram, infrastructure specifications, metric definitions, data-collection procedures, execution workflows, and aggregated before-and-after indicators. These materials support partial reproduction and analytical comparison without exposing confidential institutional information.

10 Related Work and Discussion

Prior studies have examined complementary aspects of mobile test automation. Costa and Miranda [5] compared GUI testing tools in CI environments and highlighted Appium's flexibility. Li and Cao [9] combined Appium and Selenium Grid to support execution across Android versions, while Yu et al. [19] investigated cross-platform automation using computer vision. AbuSalim [1] reported execution-time gains from parallel orchestration across Selenium Grid nodes. Table 4 summarizes these studies and their relationship to our work.

Unlike prior studies centered on tool comparison, automation techniques, or controlled parallelization, this study reports the migration of a production-oriented mobile testing infrastructure in a secure, legacy-constrained financial institution. Its contribution lies in the operational evidence, architectural decisions, and lessons learned from combining established technologies under regulatory constraints.

11 Final Remarks and Future Work

Migrating mobile test execution from Windows® to Linux using Selenium Grid and Appium improved scalability, execution efficiency, and Jenkins integration. The case study shows how established technologies can be combined with security-aware design decisions to modernize test automation in a regulated, legacy-constrained environment.

Future work will extend the solution in four directions: improving integration with the institution's legacy authentication and communication services; evaluating Podman-based runners despite the nested-virtualization requirements of Android emulators; replacing manually executed shell scripts with declarative OpenTofu configurations; and migrating the web-testing workflow to Linux. The internally developed Java testing framework will also continue to evolve to support these changes and the institution's security and operational requirements.

Another future direction is to evaluate externally managed device-farm services, such as the solution provided by Keeggo (https://keeggo.com/), as a complement or possible successor to the current emulator-based infrastructure. The evaluation will compare execution performance, device and operating-system coverage, CI/CD integration effort, availability, cost, security and compliance requirements, and vendor dependence. Its results will support the

Table 4: Comparative summary of related work and the proposed study

Study	Tools Used	Main Focus	Environment	Key Contributions	Comparison with Proposed Work
Costa & Miranda (2023)	Detox, Appium, Calabash, Maestro	Comparison of mobile GUI testing tools in CI environments	CI pipelines	Appium showed superior flexibility and integration, despite not being the fastest tool	Validates the use of Appium in complex environments with CI/CD pipelines, as done in the proposed work
Li (2023)	Appium, Selenium WebDriver	Testing across multiple Android versions	Android devices	Combination of Selenium Grid and Appium to handle fragmentation and enable parallel execution	Employs the same strategy adopted in the proposed architecture for handling OS fragmentation and concurrency
Yu *et al.* (2021)	Appium, Selenium, LIRAT	Computer vision-based test automation and cross-platform scripting	Android and iOS platforms	LIRAT framework with layout/image recognition and platform-independent scripts	Shares the goal of execution flexibility; differs by not adopting computer vision techniques
AbuSalim (2021)	Selenium Grid	Performance analysis of mobile testing with parallelization	Distributed infrastructure with multiple nodes	Parallel test orchestration across nodes significantly reduces execution time	Reinforces the performance gains observed in the proposed Linux-based distributed architecture
Proposed Work	Selenium Grid, Appium, Jenkins	Scalable mobile test automation in a banking environment	Secure network with CI/CD integration (Linux-Windows)	Successful migration from Windows® to Linux; enhanced integration with Jenkins; parallel test execution; high security compliance	

institution's decision to retain the current architecture, adopt a hybrid model, or gradually migrate mobile test execution to an external device farm.

Artifact Availability

The study artifacts cannot be made publicly available because they contain sensitive industrial information protected by a non-disclosure agreement (NDA) with the participating organization.

Acknowledgements

The authors acknowledge the support of the State Research Support Foundation for the scholarship granted through the Institutional Program for Technological and Innovation Initiation Scholarships.

The authors declare the use of the GenAI tool ChatGPT (GPT-5.5 Thinking, version 1.2026.160) during this research, in compliance with the SAST 2026 GenAI usage policy. The tool was used to assist in drafting, revising, and refining the manuscript. All content was analyzed, verified, and validated by the authors.

Maicon Bernardino is funded by CNPq (Grant #307969/2026-6).

References

[1] Samah W.G. AbuSalim, Rosziati Ibrahim, and Jahari Abdul Wahab. 2021. Comparative Analysis of Software Testing Techniques for Mobile Applications. In *Journal of Physics: Conference Series*, Vol. 1793. IOP Publishing, Kuala Lumpur, Malaysia. doi:10.1088/1742-6596/1793/1/012036

[2] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. doi:10.1002/stvr.1486

[3] Cheng-Hsiang Chiu, Zhicheng Xiong, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2024. An Efficient Task-Parallel Pipeline Programming Framework. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia 2024)*. ACM, 95–106. doi:10.1145/3635035.3635037

[4] Raiyan Rahman Chowdhury, Syeda Sumbul Hossain, Yeasir Arafat, and Bushrat Jahan Siddiqui. 2020. Configuring Appium for iOS Applications and Test Automation in Multiple Devices. In *Proceedings of the 2020 Asia Service Sciences and Software Engineering Conference* (Nagoya, Japan) *(ASSE '20)*. Association for Computing Machinery, New York, NY, USA, 63–69. doi:10.1145/3399871.3399883

[5] Gustavo Costa and Breno Miranda. 2023. A Comparative Analysis of Mobile UI Testing Frameworks in Continuous Integration Environments. In *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing* (Campo Grande, MS, Brazil) *(SAST '23)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3624032.3624047

[6] Santiago Delgado. 2008. Parallel Testing Techniques for Optimizing Test Program Execution and Reducing Test Time. In *Proceedings of the 2008 IEEE AUTOTESTCON*. IEEE, Salt Lake City, UT, USA, 439–441.

[7] Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. 2021. Flaky test detection in Android via event order exploration. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 367–378. doi:10.1145/3468264.3468584

[8] Stefan Fischer, Rudolf Ramler, Wesley K. G. Assunção, Alexander Egyed, Christian Gradl, and Sebastian Auberger. 2023. Model-based Testing for a Family of Mobile Applications: Industrial Experiences. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A* (Tokyo, Japan) *(SPLC '23)*. ACM, New York, NY, USA.

[9] Jicheng Li and Hongyu Cao. 2023. Design and Implementation of API Automation Testing System for Mobile Hybrid Mode Based on Appium Technology. In *Proceedings of the 2023 7th International Conference on Electronic Information Technology and Computer Engineering* (Xiamen, China) *(EITCE '23)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3650400.3650648

[10] Chia-Yu Lin and Shin-Jie Lee. 2023. An Efficient Autoscaling Cross-Browser Testing Cloud Platform based on Selenium Grid, Kubernetes and KEDA. *Journal of Information Science and Engineering* 39, 5 (2023), 1061–1077. http://jise.iis.sinica.edu.tw/JISESearch/pages/View/PaperView.jsf?keyId=194_2625

[11] Sundos Mojahed, Réjean Drouin, and Lokman Sboui. 2024. ODACE: An Appium-based Testing Automation Platform for Android Mobile Devices Certification. In *IEEE International Conference on Software Testing, Verification and Validation, ICST 2024 - Workshops*. IEEE, Toronto, ON, Canada, 301–308. doi:10.1109/ICSTW60967.2024.00060

[12] Maxim Mozgovoy and Evgeny Pyshkin. 2018. Mobile farm for software testing. In *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct* (Barcelona, Spain) *(MobileHCI '18)*. Association for Computing Machinery, New York, NY, USA.

[13] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 17 (Oct. 2021), 74 pages. doi:10.1145/3476105

[14] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An Empirical Analysis of UI-based Flaky Tests. arXiv:2103.02669 [cs.SE] https://arxiv.org/abs/2103.02669

[15] Per Runeson and Martin Höst. 2009. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164. doi:10.1007/s10664-008-9102-8

[16] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proc. ACM Program. Lang.* 5, OOPSLA (oct 2021).

[17] Isabel K. Villanes, Silvia M. Ascate, Josias Gomes, and Arilo Claudio Dias-Neto. 2017. What are Software Engineers asking about Android Testing on Stack Overflow?. In *Proceedings of the XXXI Brazilian Symposium on Software Engineering* (Fortaleza, CE, Brazil) *(SBES '17)*. Association for Computing Machinery, New York, NY, USA.

[18] Claes Wohlin. 2021. Case Study Research in Software Engineering—It is a Case, and it is a Study, but is it a Case Study? *Information and Software Technology* 133 (2021), 106514. doi:10.1016/j.infsof.2021.106514

[19] Shengcheng Yu, Chunrong Fang, Yexiao Yun, and Yang Feng. 2021. Layout and Image Recognition Driving Cross-Platform Automated Mobile Testing. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. IEEE. doi:10.1109/ICSE43902.2021.00139