DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2020

Evaluating the Performance and Capabilities of Popular Android Mobile Application Testing Automation Frameworks in Agile / DevOps Environment

KISHORE BAKTHA

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Evaluating the Performance and Capabilities of Popular Android Mobile Application Testing Automation Frameworks in Agile/DevOps Environment

Kishore Baktha

08/17/2020

Master's Thesis

Examiner
Mihhail Matskin

Academic Adviser
Karl Meinke

Industrial Adviser
Milena Jakovchevska

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science (EECS)
Department of ICT
SE-100 44 Stockholm, Sweden

Abstract

The number of mobile applications has increased tremendously over the last decade, thereby also having increased the importance of mobile application testing. Testing is a very crucial process in the development of a mobile application to ensure reliability and proper functionality. In an Agile/DevOps environment, test automation is an integral part in order to support the continuous integration and continuous delivery/deployment principle. There is a plethora of mobile application testing automation frameworks available in the market today. Android testing frameworks were chosen for this study because of the prevalence of Android applications in the market today. The focus of this study is the comparison of the most popular mobile application testing automation frameworks in terms of performance and capabilities in an Agile/DevOps environment. In order to achieve this, firstly, the three most popular mobile application testing automation frameworks to be used for the study were identified. Secondly, the Key Performance Indicators and capabilities to be used for comparison of the frameworks were identified. Then, test cases in a complex mobile application were designed for analysing the frameworks based on the criteria gathered. Finally, the test cases were integrated in a Continuous Integration/ Continuous Delivery pipeline to gather results and perform a more detailed comparison in an Agile/DevOps environment. From this study, Espresso turned out to be the best framework in terms of KPI analysed beating the other frameworks in all the KPIs, while Robotium was second best followed by Appium. But on the other hand, Appium was the best framework in terms of capabilities by having ability to execute the maximum number of capabilities analysed followed by Robotium while Espresso was the worst framework in this criteria.

Keywords

Mobile Applications, Mobile Application Testing Automation Frameworks, Agile/DevOps, Key Performance Indicators, Espresso, Appium, Robotium

Sammanfattning

Antalet mobilapplikationer har ökat enormt under det senaste decenniet och därmed har även vikten av mobilapplikationstest ökat. Testning är en mycket viktig process i utvecklingen av en mobilapplikation för att säkerställa tillförlitlighet och korrekt funktionalitet. I en Agile/DevOps miljö är testautomation en viktig del för att stödja principen av kontinuerlig integration och kontinuerlig leverans/driftsättning. Det finns en mängd ramverk för automatiserad testning av mobilapplikationer på marknaden idag. Testramverk för Android valdes för denna studie på grund av utbredningen av Android-applikationer på marknaden idag. Fokus för denna studie är jämförelsen av de mest populära automatiserade testramverk för mobilapplikationer med avseende på prestanda och lämplighet i en Agile/DevOps miljö. För att uppnå detta, identifierades först de tre mest populära automatiserade testramverk för mobilapplikationer som skulle användas för studien. Sedan identifierades de viktigaste resultatindikatorerna och kapaciteterna som skulle användas för att jämföra ramverken. Därefter designades testfall i en komplex mobilapplikation för att analysera ramverken baserat på de bestämda kriterierna. Slutligen integrerades testfallen i en pipeline för kontinuerlig integration /kontinuerlig leverans för att samla resultat och utföra en mer detaljerad jämförelse i en Agile/DevOps miljö. Från denna studie visade sig Espresso vara det bästa ramverket när det gäller KPI-analys, som slog de andra ramarna i alla KPI: er, medan Robotium var näst bäst följt av Appium. Men å andra sidan var Appium det bästa ramverket med avseende på kapacitet. Den hade förmågan att utföra det maximala antalet kapacitet som analyserades, följt av Robotium medan Espresso var det sämsta ramverket i dessa kriterier.

Nyckelord

Mobilapplikationer, Mobilapplikation Testar Automatiseringsramar, Agile/DevOps, Viktiga Resultatindikatorer, Espresso, Appium, Robotium

Acknowledgements

I would like to thank the company Truecaller for giving me the freedom to work on this thesis as well as all my colleagues for their guidance whenever I got stuck during the implementation of this study.

I would like to thank my industrial adviser Milena Jakovchevska for her constant support, motivation and guidance throughout the completion of this project. I would also like to thank her for giving me the opportunity to work on this thesis as it was her who had this idea in mind and let me work on it.

I would also like to thank my thesis supervisor Karl Meinke for the informative feedback throughout this project by giving me great ideas on how to work on this project and making this project as best as it could be.

I would like to thank my examiner Mihhail Matskin for agreeing to grade this study.

Finally, thanks to my family for everything and words alone cannot explain how grateful I am to have you. Last, thanks to the Almighty who always makes me believe, “there is a light at the end of the tunnel”.

Stockholm, August 2020
Kishore Baktha

Table of Contents

List of Figures

List of Tables

List of acronyms and abbreviations

API	Application Programming Interface
APK	Android Package
AUT	Application Under Test
CD	Continuous Delivery
CI	Continuous Integration
CIViT	Continuous Integration Visualization Technique
CLOC	Count Lines of Code
CPU	Central Processing Unit
CRTS	Continuous Regression Test Selection
CTSP	Continuous Test Suite Prioritization
DevOps	Development and Operations
DIA	Distance to Ideal Alternative
GUI	Graphical User Interface
HTML	Hypertext Markup Language
KPI	Key Performance Indicator
LOC	Lines Of Code
MADM	Multiple Attribute Decision Making

mAh	Milliampere Hour
OCR	Optical Character Recognition
OS	Operating System
RQ	Research Question
SDK	Software Development Kit
SSH	Secure Shell
XML	Extensible Markup Language
UI	User Interface
USB	Universal Serial Bus
VCS	Version Control Systems
VoIP	Voice Over Internet Protocol

1 Introduction

This chapter introduces the problem that this thesis addresses, along with the context of the problem and the goals of this thesis project. Also, the structure of the thesis is outlined along with the research questions to be answered.

1.1 Background

Since the introduction of smartphones, there has been a steady growth in the number of mobile applications. Almost every individual in the world has a smartphone today to serve daily purposes. Mobile apps can be used for various purposes such as finding the location of the user, listening to music, sending messages, etc.

The most commonly used OS for smartphones in the world to this date is Android. Google Play store is the primary distributor for Android apps. It is estimated that there are currently 2.9 million apps in the Google Play Store [1].

Almost every day, an Android app is released to the Google Play Store or is updated. While there is constant demand for developing new apps, there is also an equal necessity for the stability and reliability of the app. If there is less importance given to testing than with development, the app may become unresponsive or crash regularly. Thus, major mobile app companies now have testers who are responsible for testing the functionality of the mobile app. This type of testing is called manual testing where the testers test each feature of the app manually by hand and verify whether the required functionality is satisfied.

Mobile Application testing automation is the practice of using test automation frameworks to automate testing certain functionalities of the mobile application. The frameworks are responsible for test case generation, execution and then verification. Regression testing is a type of testing wherein the functional and non-functional tests are re-executed in order to verify that the previously developed software still works as expected after a change. Implementing test automation will allow the organizations to run regression testing without wasting the effort of humans to verify the same features again when they can focus more on verifying the latest developed features.

Many software organizations have complex and evolving requirements due to which the traditional software methodology of practicing the waterfall model cannot be used anymore owing to the strong rigidity of the model. Thus, the organizations are moving to Agile/DevOps environment for faster software development. Agile focuses on iterative and incremental development of the software while DevOps focuses on incremental deployment of the software.

There have been many studies that have focused on comparison of mobile application testing automation frameworks. In [2], information about different test automation frameworks were mentioned in terms of difficulties, the operating system supported and testing types. In [3], the authors have done research about the existing studies that are related to testing of Android applications. They have found that there is lack of research of comparison of test automation frameworks with respect to performance and most of the studies that mention comparison have not been done on an industry setting and therefore it would be difficult to decide which framework to actually use in the practical world setting. Moreover, the authors in [3] have also found that there are very few studies that mention the software testing metrics that can be used to evaluate the different mobile application testing automation frameworks.

1.2 Problem Statement

Performing regression testing manually in an Agile/DevOps environment can be very tedious and time-consuming owing to the continuous change in the software being developed leading to the requirement of testing the app from scratch for every new feature being developed. Thus, it is necessary to implement test automation to address the rapid change of software. There are certain challenges that needs to be addressed while implementing test automation in Agile/DevOps environment. A major factor that affects the performance of test automation is the selection of test automation frameworks. Many software companies have developed different frameworks with different capabilities and

characteristics. The problem is to address the challenges of mobile application testing automation in an Agile/DevOps environment and then implement and perform a comparison of the popular mobile application testing automation frameworks in such an environment.

1.3 Purpose

The process of performing regression test manually has become very tedious and time-consuming due to the increase in practice of Agile/DevOps environment, where new features are integrated to the mobile application every day. A lot of time and resources is wasted in verifying whether the old features work again and again. Thus, the purpose of this project is to conduct research on mobile test automation and it's usage in an agile/DevOps environment to reduce both the cost and time of manual testing. The study is performed using the most popular test automation frameworks on a complex mobile application currently in the market to get more accurate results.

1.4 Goals

The goal of this project is to perform comparison of popular mobile application testing automation frameworks in an Agile/DevOps environment. This has been divided into the following four sub-goals:

1) To identify the KPIs to be used for performing comparison among the frameworks.
2) To identify the mobile application testing automation frameworks in market today and select the most popular ones for comparison.
3) To identify the challenges of integrating mobile application testing automation frameworks in Agile/DevOps environment and propose mechanisms to address them.
4) To integrate the most popular mobile application testing automation frameworks into Agile/DevOps environment and perform comparison.

1.5 Research Methodology

The research methodology used here is a *Quantitative Research* as it involves experiments of collecting data, measuring variables and falsify or validate hypothesis or thesis. In this study, the quantitative research is observing the values of KPI metrics obtained in different frameworks and then defining a metric whether the value obtained ranks the framework higher or lower compared to other frameworks.

As a research method, we believe that *experimental* is more suitable for investigating the variables and evaluating the performance of different test automation frameworks. This method is implemented by keeping one or more variables constant and then manipulating the other variables to observe how the changes affect the performance of the mobile application under test.

Along with the quantitative methods, we will use *deductive approach* which involves testing theories with the help of large data sets. An *experimental strategy* will also be used with the help of statistics to compare the performance of test automation frameworks. *Exploratory data analysis* will be conducted using the data sets gathered to study and compare different frameworks and come to a conclusion on which framework is better than the other.

In order to qualify the results of this project, the experiment will be conducted multiple times using different test cases to perform a suitable comparison between the different frameworks.

1.6 Ethics and Sustainability

During the execution of the entire research project, only internal phone numbers were used. The data was collected purely from the measurements used in the code such as the execution time and the report generation frameworks. The data collected is confidential and is not leaked out in any manner to third party entities. It is kept by the company - Truecaller, where this project was conducted in. The data will be kept as long as the results can be used in any matter to further enhance the project carried out.

No participants were involved in this research and no user data was collected in any manner. Efforts were taken to preserve the power usage by using the mobile devices and the computer connected to this experiment only when needed.

1.7 Delimitations

The project compares three of the most popular mobile application testing automation frameworks. It can be extended in the future for comparing more mobile application testing automation frameworks. The test cases to be implemented in this project are carried out on one complex mobile application and can be extended to include more complex mobile applications.

1.8 Structure of the thesis

The thesis is structured as follows:

- In chapter 2, a background of technology used – Types of Testing, Android App Development, Mobile Application Testing, Mobile Application GUI Testing Automation and Agile/DevOps environment are mentioned. Also, the literature review conducted in this study is presented.

- In chapter 3, the research methodology used is discussed along with the setting up of the test environment.

- In chapter 4, the methods used for conducting this study are elaborated with the implementation of the KPIs, implementation of the CI/CD pipeline and addressing the challenges of integrating mobile application testing automation frameworks into the pipeline.

- In chapter 5, the results from the test runs along with preliminary analysis is presented.

- In chapter 6, discussion is done based on the results obtained from the test runs.

- In chapter 7, the conclusions are presented along with limitations and future work.

2 Background

This chapter gives a summary of the overall background information relevant to this thesis project.

2.1 Background Technology

2.1.1 Types of Testing

The different types of testing that are relevant to this study are:

1. Black-Box Testing – This is a type of testing where the tester does not need to have knowledge about the internal design/structure of the project being tested [4]. The tester does not need to know the source code and testing is done based on inputs given to the system which acts like a black box. The general process in black-box testing is that the tester is given a set of requirement specifications. This is converted to a set of test specifications and then the system is tested by giving inputs and verifying whether the output is the expected result or not. It can be used to find errors such as Interface errors, Behaviour errors or Termination Errors. The main advantage is that the tester works independently of the application programmer and does not need to have knowledge about how the software is implemented. The main drawback is that only a small set of possible inputs can be used for testing and different program paths might be left untested.
2. White-Box Testing - This is a type of testing that takes into account the internal structure of the system being tested. The design of the test cases is based on the implementation of the software entity [4]. This type of testing is often done by the developer of the code but sometimes could also be done by the tester who has access to the source code. The main goal is to generate inputs for testing such that all the execution paths in the program are traversed. It helps in the detection of logical errors in the code. The main advantage is that the entire code used for software development can be tested for errors and also checked whether proper code style is followed. The main drawback is that it adds additional complexity to testing because the tester needs to have knowledge of the system being tested.
3. Gray-Box Testing – This is a type of testing used to test the system with partial knowledge of the internal structure of the system. It can be used to identify defects due to improper code structure or improper functioning of the system. It gives the possibility of testing the presentation layer of the system as well as the code. The process of testing is similar to that of black-box testing with additional inputs covering more program paths. The main advantage is that it provides the combined benefits of both black-box testing and white-box testing. The main drawback is that there is a possibility of duplication of test cases by the tester and the programmer.
4. Functional Testing – It can be described as black-box testing of individual components of the system to test each and every function for ensuring their proper behaviour. It involves identifying the program's features and variables used by the functions and then test the behaviour of the functions. Each function is initially tested with a small set of inputs which are then expanded as much as possible. The main purpose is to check the functionality as well as usability of the application. It is also useful for checking any error conditions and whether proper error messages are displayed.

2.1.2 Android App Development

Android is one of the most commonly used OS of a mobile device. Android app development refers to the process of creating mobile apps that will be running on the Android platform.

The most common components of Android apps are [5]:

1. Activities - They are used to build the UI of the mobile application. They are primarily the different screens observed and the design of each screen is represented by a XML file.
2. Services - These are basically components that run without the UI and are responsible for performing functions such as updating the data source, triggering notifications, etc. Normally, they execute in the background.

3. Intents - They are basically used while moving across different screens and also for transferring data.
4. Broadcast Receivers - They are responsible for performing some action when a specific event takes place. For example, they could be used for sending text messages at specific actions.
5. Content Providers - They are primarily used to share data across application boundaries by allowing other applications to read current application's data and vice-versa.

The app is run on the mobile device with the help of APK. It contains all the required components for running a mobile app.

2.1.3 Mobile Application Testing

Mobile Application testing refers to the process of testing the quality of the mobile app with respect to identifying defects in order to meet user requirements and improve user satisfaction and experience. There are several factors specific to mobile applications that need to be kept in mind while testing them. They are [6]:

1. Mobile Connectivity - It is one of the most crucial characteristics of a mobile application. The mobile application may connect to a network that might vary in speed, security and reliability. The mobile application's correct functioning may strongly depend on the type of network especially when the data needs to be fetched from a network resource.
2. Limited Resources - The limited resources available to a mobile application may affect the functionality due to performance degradation and while testing, this scenario needs to be taken into account.
3. Autonomy - This basically refers to the energy consumption in a mobile device which may vary in different devices depending on the battery capability. Hence, while testing a mobile application, the energy consumption of the application in the device should also be evaluated.
4. Context Awareness - Mobile apps may rely heavily on sensed data such as noise, temperature, etc. The sensors in the devices may produce different outputs depending on the environment conditions. Testing should be done in as many environments as possible.
5. Touch Screens – These are the main input source for mobile applications. Hence, it is necessary to have testing techniques to test the functioning of the touch screen in different scenarios.

2.1.4 Mobile Application GUI Testing Automation

Mobile Application GUI testing automation is the process by which the mobile application testing automation framework executes the testing by automatically performing certain functionalities based on the test suite written by the user and then finally verifying whether the result obtained is the same as the expected result. The implementation of the functionalities is done by the test automation framework selected by the user. The expected behaviour of the system is validated through execution of the test cases.

The test cases can be run on either the emulator or a real mobile device but running the test case in a real device is preferable, as emulators are not actual devices and the actual test results may differ if run in an actual device due to various factors such as context awareness as discussed above.

Different test automation frameworks will have different mechanisms to run the test suites but the end result obtained is the same for all the frameworks. The user does not manually have to implement the steps in test case and has to only check whether the test cases have passed or failed. If the test case has failed, the user can observe the reason through different techniques such as screenshots or logging and then report the bug to the concerned authority.

The main goal of GUI testing is to ensure correct behaviour and state of the GUI [7]. The verification of the GUI testing can be done by bitmap comparison, using an API or by optical character recognition. There are several challenges associated with GUI testing. There could be several states of the GUI and it would be impossible to test all of them as the user can navigate through several paths. The event-driven nature of GUI makes GUI applications non-deterministic and it might be highly error-prone [7]. The tests might also fail even though the state of the application is correct due to external factors such as notifications in the mobile device.

2.1.5 Agile/ DevOps environment

The rapid change in requirements of the software and the demand for faster development and deployment of the software have made organizations switch to Agile/DevOps environment.

Agile emphasizes continuous integration (CI) principles. CI involves the use of CI servers such as Jenkins to automate the build process. The developer pushes the new code to the CI server using a VCS such as Git after testing the code locally. The CI server now merges the new code with the latest version on the server [8]. If there are conflicts, then the developer has to resolve them before pushing merging the code. This process can be repeated many times during the day by the same developer or different developers working in an organization.

While Agile focuses on automating the development process, DevOps goes a step further and also focuses on automating the release process. It emphasizes the CD principle. CD involves the design of infrastructure for automating the deployment and release of the software once the software is integrated to the CI server. It is up to the users to decide the frequency of the release. CD can be used as Continuous Delivery or Continuous Deployment. The difference is that in Continuous Delivery, the process of releasing the latest version of the app into production is carried out manually while in Continuous Deployment, the process of release is automated. Continuous Delivery can be used in all kinds of software organizations while Continuous Deployment may be suitable only for certain types of organizations.

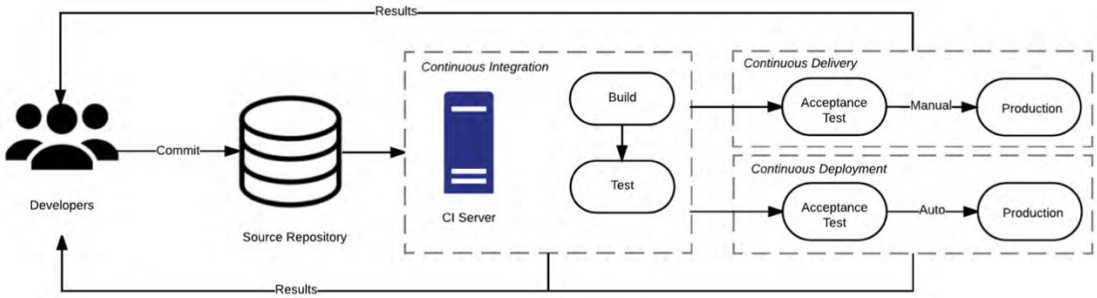


Figure 2.1: Relationship between continuous integration, delivery and deployment [9]

Organizations that follow both Agile and DevOps generally make use of the CI/CD pipeline to integrate both practices into their software lifecycle.

2.2 Related Work

In this section, the literature review is presented which consists of answering part one of the research questions.

2.2.1 Previous Study on Mobile Application Testing Automation Frameworks (RQ 1.1)

Since the performance of test automation strongly depends on the frameworks selected, it is imperative to select the right framework based on the features and requirements of the app to achieve the best performance.

There are several papers that have been published that mention mobile application testing automation frameworks. In [10], three frameworks - Appium, Espresso and Calabash were mentioned as the most commonly used frameworks for functional UI testing. Calabash was found to be the best one followed by Espresso and then Appium. The result was based on execution time and element inspection mainly for performance evaluation. The DIA algorithm which is part of the Multiple Attribute Decision Making (MADM) aims to select the best alternative while ensuring no ranking abnormalities [10] and was used for performance evaluation in the above research paper.

Appium, Robotium, UiAutomator were the frameworks mentioned as the most used and were compared in [11]. Appium was found to be able to handle different functional testing scenarios such as switching network mode, handling interruptions, opening other applications, etc. UiAutomator similarly can handle all mentioned scenarios but it cannot handle opening other applications. Robotium was found to handle them partly. In terms of execution, UI Automator was found to have least execution time.

In another study [12], many frameworks, but mainly - Appium, Robotium, MonkeyRunner and Calabash were discussed. Basic information about frameworks was given with Appium shown to have more functionality such as support for built-in applications such as camera, calendar, phone. MonkeyRunner seems to be available for functional testing but writing different test cases for every device is not a good option considering the plethora of devices in Android. Types of testing such as Unit Testing, Integration and Security was discussed along with the challenges which test automation frameworks have for supporting them.

Espresso and Eye-Automate frameworks were compared in [13]. While comparing the frameworks, it was stated, based on a study of survey of participants involved, that Espresso involved less failures due to direct testing of UI components and also Espresso was able to test at a lower level than Eye-Automate. In paper [14], many frameworks were mentioned with Espresso, MonkeyRunner and Robotium quite popular. The authors have performed a case study of automation functional testing in mobile applications and have found out that MonkeyRunner is used in a majority of studies for comparison as it can be used without almost no configuration effort.

Apart from research papers, there were also some blogs studied that mentioned mobile application testing automation frameworks. Appium, Espresso, Calabash, Selendroid and Robotium were mentioned in [15]. Appium was shown to be a very popular framework with the capability to support cross-platform (Android and iOS). Espresso was mentioned as a widely used framework with the capability of record and playback feature. In [16], Espresso, UIAutomator, Appium were studied and different pros and cons of each framework were given. Appium had an advantage of being a cross-language framework but the drawback is that execution of test cases is slow. Espresso can execute test cases very quickly but test cases can only be written in Java or Kotlin. UiAutomator offers interaction with system components but there is no support for web elements.

2.2.2 Detailed Description of Common Mobile Application Testing Automation Frameworks

Appium - This was found to be studied in most of the papers and blogs. Appium is very similar to Selendroid used for the mobile web application test automation and makes use of web drivers. It is a black-box testing framework and therefore, the testers need not to have access to the source code of the mobile application while performing test automation. It makes use of the inbuilt UiAutomator framework for the Android platform. It can test any type of mobile application - mobile, web and hybrid. It is open-source and provides cross-platform and cross-language capability (can be written in many coding languages such as Java, Python, Ruby). The minimum Android SDK version required is 16.

Espresso - This is a framework provided by Google that is mainly used for functional UI testing of the AUT. It requires the knowledge and access to the source code to automate test cases. The testing code and the source code are kept in the same place. The framework has the ability to identify elements used in the application directly thereby resulting in faster execution of test cases. It has a dependency on Hamcrest library and uses matchers from it for finding and performing some action on elements [16].

Robotium - This framework can be used for writing UI functional tests. It can also only be used to navigate inside components in an AUT like Espresso as it strongly depends on the instrumentation framework [17]. It supports gray-box testing of the Android apps which is a combination of black-box and white-box testing. It is open-source with a dedicated developer community. It supports both native and hybrid mobile applications.

MonkeyRunner - This can be used to test a mobile application at different levels, such as function or framework as well as unit test cases [12]. It has features to allow the multiple test suites to run in multiple devices simultaneously. It is supported by Google and is used for sending inputs to the AUT and taking screenshots. It can be used for testing at a lower level than Espresso and can run on any Android version. It is a black-box testing framework.

Calabash - This is a cross-platform testing framework having support for both native and hybrid mobile applications. It is mainly used with the cucumber framework to execute test cases and supports many gestures and assertions [17]. It is a black-box testing framework and supports multiple coding languages - Java, Ruby, .NET and Flex.

The Table 2.1 summarizes the above frameworks based on different categories.

Table 2.1: Mobile Application Testing Automation frameworks

Framework	Type of Testing (black, white, gray)	Coding Language	Platform Support (Native, Hybrid, Web)
Appium	Black	Java, C#, Python, Ruby, JavaScript	Native, Hybrid, Web
Espresso	White	Java, Kotlin	Native
Robotium	Gray	Java	Native, Hybrid
MonkeyRunner	Black	Jython(Python using Java)	Native
Calabash	Black	Java,Rudy,.NET, Flex	Native, Hybrid

2.2.3 Popularity of Mobile Application Testing Automation Frameworks (RQ.1.2)

Though mobile application testing automation frameworks have been discussed extensively in many papers, there have been no papers found by the author that discusses the popularity of these frameworks. Thus, in order to get information about the current most popular frameworks, Google Trends [18] which gives information about the relative Google searches and Stack overflow [19] which is a website where developers post and answer questions were used.

2.2.3.1 Popularity based on Google Trends

Google is one of the most used platforms to search for any information. Through the use of Google Trends website, it is possible to get information about the relative number of searches done by users all over the world on particular terms. It also allows one to perform comparison among different terms and the results are generated in the form of graphs.

In order to get the popularity of different mobile application testing automation frameworks, the common mentioned test automation frameworks in different papers were entered as terms in Google Trends website. Based on Table 2.1, the five commonly mentioned test automation frameworks were used and the keywords were the framework name appended with “android” in order to get more accurate results.

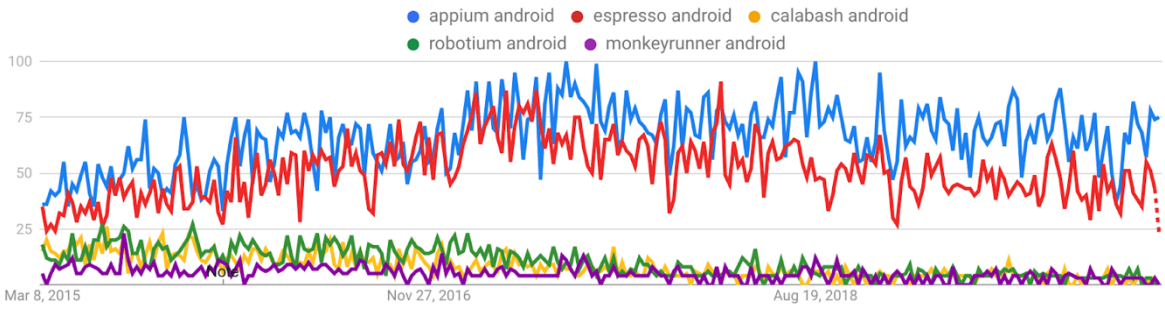


Figure 2.2: Number of Google searches in terms of relative percentage

Based on Figure 2.2, it can be observed that Appium seems to be the current most popular framework followed by Espresso, in terms of Google searches. The third place is very close among the frameworks Robotium, MonkeyRunner and Calabash.

2.2.3.2 Popularity based on Stack overflow

Stack overflow is one of the most commonly used websites for answering any software development questions. It is widely used by developers around the world almost every day. To obtain the most popular frameworks, the framework names were passed as tags and a number of questions in Stack overflow having the same tag name were observed. The Stack overflow website dataset was obtained through the data explorer site called stackExchange [20]. The framework names were passed as tag parameters in the query to obtain the result.

Table 2.2: Tags and Count of Questions on Stack overflow in March 2020

Tag Name	**Count of Questions**
Appium	6031
Android-espresso	2831
Robotium	1007
Monkeyrunner	592
Calabash	471

Based on the Table above, it can be observed that Appium, Espresso and Robotium are currently the most popular frameworks in terms of most questions asked in Stack overflow followed by Monkeyrunner and Calabash.

The scope of this thesis project will be the comparison of the three most popular frameworks – Espresso, Appium and Robotium.

2.2.4 Architecture of the most popular frameworks

The architectures of the three most popular frameworks – Appium, Espresso and Robotium are presented below.

Appium

To execute test cases using Appium, it is required that the app is either already installed in the device or the path to the APK can be provided when the Appium server is started where the installation of the app takes place before the test cases are run. In the case of Android, Appium uses UiAutomator or Selendroid in the background to execute the test cases in the device.

Appium follows a client-server architecture. Initially, a web server has to be started on a specific port that can listen to client connections. A client can connect to the server using the port and then it can send commands to the server. Communication between the client and server is in the form of response and requests. The client sends requests for performing some kind of automation to the server. The server then processes the requests and responds with the test result or log files.

The communication between client and server takes place within a session that is initiated by the client. The client is also required to send a set of JSON objects called “Desired Capabilities” to indicate different settings under which the session has to be started such as the ‘platformName’ . The main reason for this is to allow the framework to distinguish between the platforms – Android and iOS since it works differently on both the platforms. The session communication takes place with the help of JSON wire protocol. This protocol is a set of standardised endpoints that the client is exposed to for allowing the webdriver to establish connection between the client and server for performing automation [21].

The Appium server is written in Node.js. The execution of Appium in the Android device can be summarized in Figure 2.3. The Appium client (using libraries) first connects with the Appium server using JSON wire protocol for communication. The Appium server creates a session and verifying the “Desired Capabilities”, connects with the corresponding vendor-specific frameworks like UiAutomator.

The UiAutomator now communicates with bootstrap.jar that is running in the emulator or real device. The bootstrap.jar plays the role of TCP server which can be used to send the test command to perform an action on the Android device using UiAutomator.

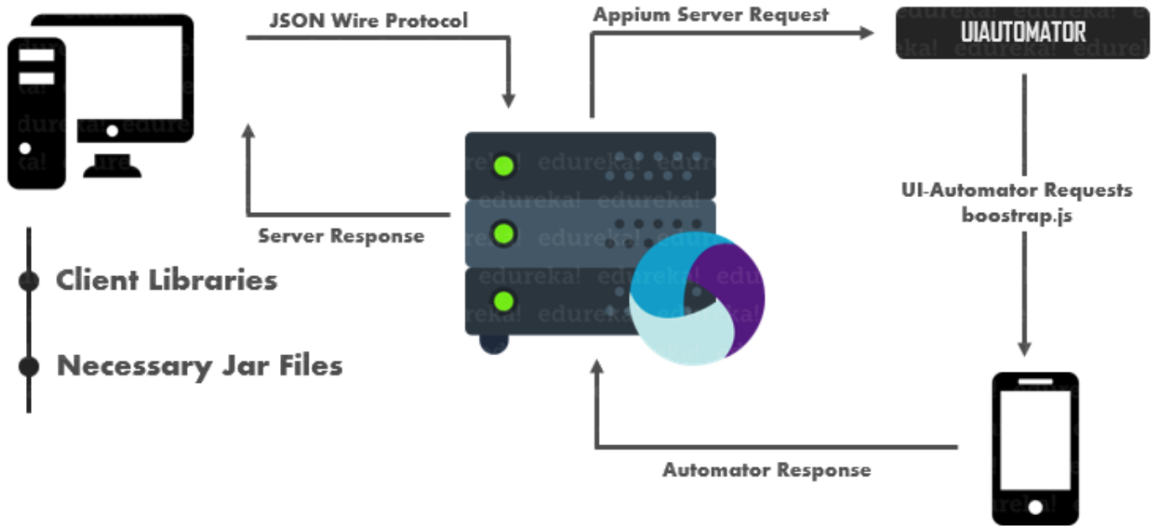


Figure 2.3: Appium architecture [21]

During the execution of the tests, an instance of AndroidDriver is initialized which is used to communicate with the device. The instance can be used to fetch objects from the device and interact with them for performing various actions. For example, to click a button , the command used is:

***driver**.findElementById("com.example.debug:id/button_reject_call").click();*

Espresso

Espresso can be imported as an library to the existing Android project. It uses AndroidJUnitRunner which is basically used for running Junit test cases. The library is responsible for installing the AUT to the connected device or emulator and also loading and running the test package. The code written for the test cases is directly included in the code of the application and hence they can access all the elements and corresponding classes directly. It is based on the instrumentation framework so the test application runs in the same device as the AUT.

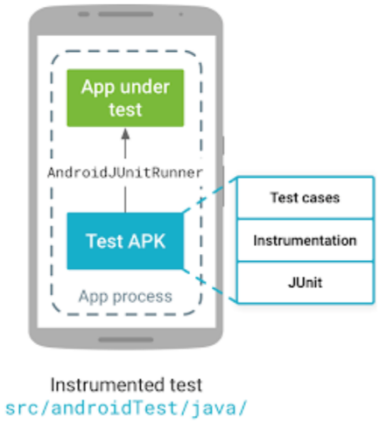


Figure 2.4: Instrumentation test on Android [22]

Espresso provides automatic synchronization of test actions with the UI of the AUT. It detects when the main thread is idle to run the tests at the appropriate time. This improves the reliability of the test as only the main thread can do any updates to the UI widgets and therefore, Espresso makes sure the application is an ideal state before test execution [23].

The framework needs to know which activity in the application to run and this is specified using the @Rule annotation. The annotation can also be used for enabling permissions for the application before starting the activity. The test cases are run with the @Test annotation. Espresso can be used together with other frameworks such as UiAutomator through the addition of more libraries. For execution of the tests, the static methods of the Espresso framework are called. For example, the command used to click a button is:

*onView(withId(R.id.**saveButton**)).perform(click());*

Robotium

Robotium is the oldest testing framework compared to the other two frameworks used in this study. It is a very low-level framework mainly developed to test a single activity in an application. It is widely used for functional UI testing but can be used for system and acceptance tests as well. It supports both native and hybrid application tests.

Robotium is also an instrumentation framework like Espresso and thereby runs similarly, with the test application running in the same device as the AUT. Like Espresso, Robotium can also access the elements and corresponding classes directly and runs using AndroidJUnitRunner. It has the same architecture as shown in Figure 2.4. The running of the test cases are similar to Espresso with the usage of annotations.

The main difference between Espresso and Robotium is that Robotium extends an outdated ActivityInstrumentationTestCase2 and Robotium does not have automatic synchronization like Espresso. For test execution, Robotium uses Solo class for interacting with the device. The Solo class is initialized by passing an instance of the instrumentation framework and then the methods of the class are called for performing various actions. A Button can be clicked using the command:

***solo**.clickOnView(**solo**.getView(R.id.**nextButton**));*

2.2.5 KPIs for Mobile Application Testing Automation Frameworks (RQ 1.3)

In order to perform a suitable comparison among different mobile application testing automation frameworks, it is necessary to identify the most suitable KPIs that can be used for measuring the effectiveness of frameworks. The following are some of the most useful KPIs -

1. **Execution Time -** This is the one of the main factors that can be used for comparing the performance of two different frameworks. It refers to the time taken for all the automatic tests to run to completion. The execution time can be dependent on various factors such as time for loading to a device, time for locating the elements [12]. Ideally, a better performing framework performing better should have less execution time. It is necessary to design the test cases in such a way that execution time is solely based on the framework capability and not on external factors such as network latency, battery shortage, etc. Moreover, execution time is also an important factor in an Agile/DevOps environment as longer time may affect the speed of continuous integration and deployment of software due to the delay in test cases getting passed.

   The capability of the framework plays a strong role in the total execution time of the test suites. For example, if the framework can execute tests in parallel, it will greatly reduce the execution time. For this, the tests have to be independent of each other. Also execution time can be reduced if the framework has capabilities to automatically detect functions such as incoming calls rather than wait until the call element is identified [17].

2. **Maintainability -** This is concerned with the number of LOC to achieve the same test suite in different frameworks. If the framework can execute the same functionality in fewer numbers of

lines, then it's believed to be more maintainable than the framework requiring more lines. Test fragility is the frequent need for maintaining test cases due to the change in the GUI of the app [24]. Fragility can be a significant problem as a failing test case might require in-depth investigation until the problem is found [25]. There are different types of fragility depending on the changes made in the app such as change in text or change in application flow. The changes in LOC due to test fragility also gives an indication of the maintainability of the framework.

3. **Flakiness –** Test Flakiness is defined as the failure of test cases due to non-deterministic outcomes leading to some executions of a test to end with failure, but not all [26]. In complex apps, this could occur due to high unreliability of the back-end service or network connection. In terms of mobile application test automation frameworks, it could occur due to some frameworks taking longer time to detect elements in some cases. In order to measure the degree of different frameworks to handle flaky tests, the number of waits inserted in the code can be calculated. This number basically refers to the number of wait statements of the framework used. Usage of waits will avoid most of the tests from failing if the time of wait is enough for the framework to detect the elements. Test flakiness can also depend on various other factors such as automatic triggering of garbage collection in Android apps because the device is low in memory [27]. Different frameworks may have a different degree of flakiness and a better framework should have smaller number of waits. High memory usage in the device can also cause lag in the GUI leading to more flaky tests [24]. It is to be noted that sometimes, the test can be flaky due to a behaviour of the app such as a new popup that might hover around the element to be detected causing the test to fail. However, since such situations do not give a suitable comparison between frameworks, it was neglected from this study.

4. **Reliability -** The reliability of a framework can be measured as the number of test cases that fail when they shouldn't. This can be measured as the framework being incapable of passing tests due to reasons that are solely framework specific. For example, some functionality in the framework such as opening the navigation drawer or scrolling could be unreliable. Some frameworks might not be able to detect the app in the mobile device causing tests to fail. Also, some frameworks may fail the tests due to the framework drivers not being able to be instantiated. The inspection of tests that fail due to some reason not related to code defect can result in a wastage of resources [28]. A better framework should be more reliable in terms of fewer failures due to framework defects rather than actual bugs in the mobile application.

5. **Fragmentation -** Different mobile devices may have different configurations such as different OS, Android version, etc. [27]. This is called fragmentation and has made the testing of Android apps very difficult. The different types of configuration can affect the tests resulting in pass or fail and some kind of effort will be required in terms of code change or even the way elements are detected using frameworks in order to try and make the test pass in as many devices as possible. This can be a good measurement to determine the performance of frameworks. Ideally, a framework should require few changes in the code and modifications in the way elements are detected to run successfully in multiple devices.

6. **Identification of defects –** Another KPI indicator is the ability of a framework to identify defects or bugs in the mobile application that have not been detected during manual testing [29]. This strongly depends on the test cases selected and the framework's capability to identify edge cases that have not been noticed before. Some frameworks may execute the tests very quickly and may be less reliable and not be able to identify the bugs but other frameworks may execute the tests slowly and during such executions, a bug could be identified. Hence, it would be interesting to see which framework identifies bugs with the same test cases. However, there is no guarantee that the bugs can be identified by any framework as the main purpose of the test automation frameworks is to detect regression bugs in existing functionalities that have not been found before.

7. **Battery Usage -** Identifying the battery consumption in mobile device for different frameworks can be used as a KPI measurement. Battery consumption is the drain in battery due to various usages such as power, memory or network usage. Memory usage refers to the CPU resources required by the framework for running the test suite in different devices [30] which causes drain in battery. Power usage is measured as the charge consumed when testing the application. Network usage refers to the Wi-Fi usage in the mobile device causing battery

drain. High resource consumption could cause the mobile devices to lag and maybe even cause the tests to fail in certain situations. A framework having better performance should have less battery consumption for the same test suite.

2.2.6 Challenges of Integrating Mobile Application Testing Automation Frameworks to Agile/DevOps environment(RQ 1.4)

Due to the high frequency of development and release in the Agile/DevOps environment, there are certain challenges associated that need to be discussed and handled while integrating mobile application testing automation frameworks into such an environment. Some of the challenges are :

1. **Unstable test cases -** Sometimes the tests may be unstable and fail irrespective of the code [31]. This can happen because of the unpredictable nature of the functionality being tested and this could seriously affect the CI principle as new code will not be allowed to integrate due to failure of test cases. Moreover, there will also be a wastage of time in identifying whether the failure of the test is due to a bug or false positive/negative. Thus, maintenance of the automation test suites due to unstable test cases is a big concern especially in an Agile/DevOps environment.

2. **Lack of proper infrastructure -** To follow the practices of Agile and DevOps, a CI/CD pipeline is used. It is necessary to have a proper infrastructure designed to integrate the test suite into the pipeline. A suitable CI tool should be used and sometimes changes may have to be made with respect to the configuration of the CI tool for supporting integration of test suites. Another problem is also the security and reliability issues associated with the tool used especially during the release process which might lead to increased errors and delays affecting the continuous delivery practice [9].

3. **Distributed Organization -** When there is a large scale organization with multiple teams, if each team does the implementation of automation test suites separately and then integrates to a separate server, then there might be integration issues when everything has to be merged [9]. This mainly occurs in an Agile/DevOps environment when there are multiple scrum teams, each implementing their own test suites. It could also lead to various other issues such as duplication of test suites resulting in wastage of time and effort.

4. **Build Time -** Dependencies among test cases can increase build time enormously affecting the performance and efficiency of CI [9]. This basically refers to the feedback loops from the test cases. The ideal approach is to isolate the dependencies by making independent tests to run in different processes. Different techniques such CRTS and CTSP could be used to decide which test suites to execute at what intervals. Moreover, builds associated with CI systems requires extra computation, bandwidth and memory resources and efforts should be made to reduce the extra resources [9].

5. **Awareness and Visibility of Build and Test Results –** As more code integration happens, the information corresponding to build and test results would increase exponentially [9]. This could slow down the feedback in CI as it would take more time to interpret the results generated from the build and make the necessary changes required. Therefore, an appropriate strategy needs to be decided in the selection and usage of tools in order to present the results in a more simplified and interpretable manner. Techniques such as CIViT could be used to represent the testing efforts intuitively [32].

3. Research Methodology

This chapter presents an overview of the research methodology used in this thesis.

3.1 Research Process

The research process is presented in Figure 3.1. The project started with the literature review. This was done by going through relevant blogs and scientific articles/papers. Through the literature review, Part one of our research questions could be answered. First, an identification of the mobile application testing automation frameworks available in the market today was carried out. Next, the popularity of the frameworks was researched. Then, KPIs that could be used for comparison of frameworks were identified. Finally, the challenges of integrating frameworks to Agile/DevOps environment were identified.

The second part of the study involved performing empirical analysis on the results obtained. This was done by firstly, designing and implementing the test cases on the selected frameworks. Then, the design and implementation of the pipeline was carried out for integrating the frameworks into the Agile/DevOps environment. Next, the methods to measure the KPIs were decided and implemented to obtain results. Finally, the frameworks were evaluated based on different criteria.

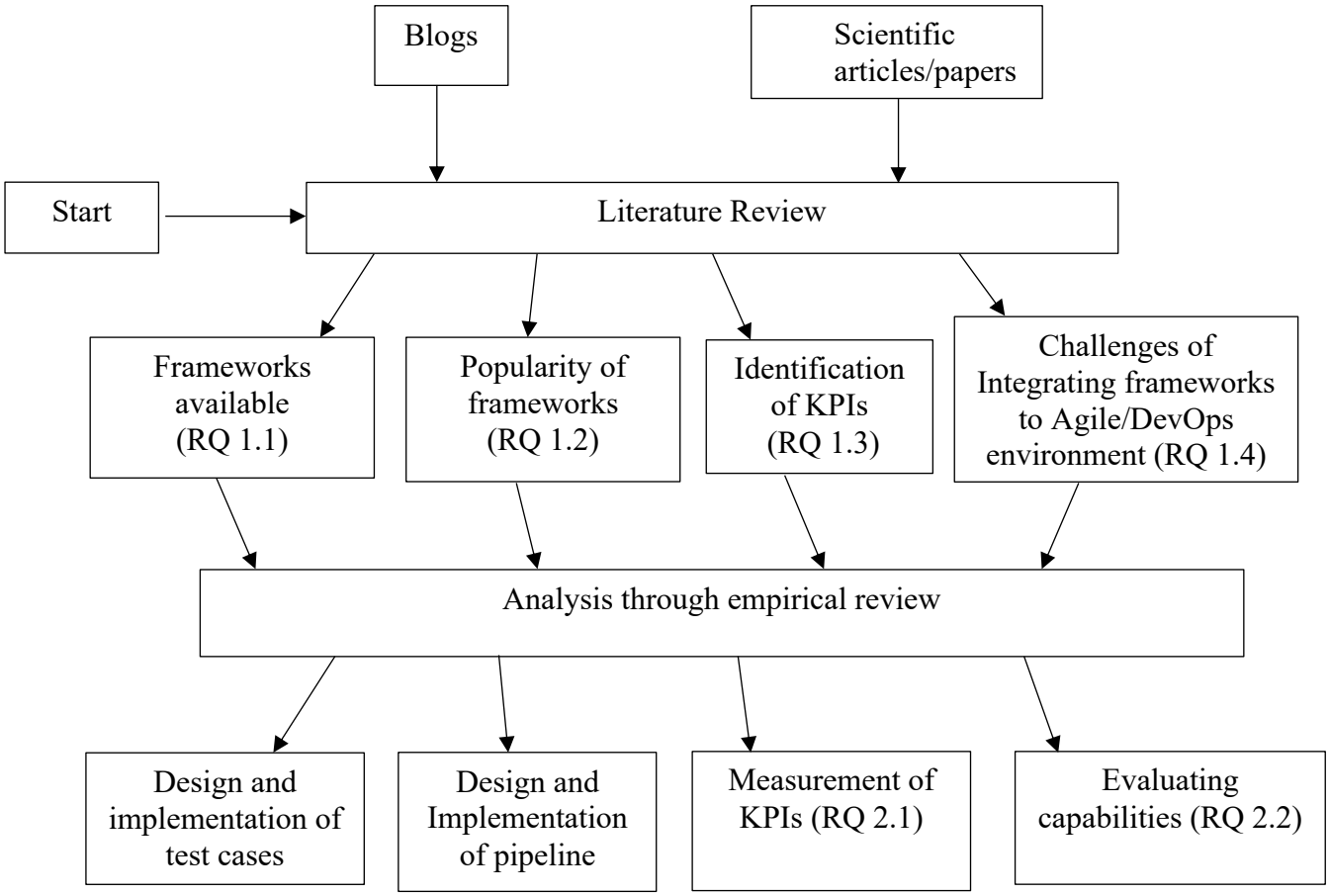


Figure 3.1: Research Methodology

3.2 Research Questions

The research questions answered in this study are –

RQ 1.1 : What are the frameworks available for mobile application test automation?

RQ 1.2: How do the frameworks rank by popularity?

RQ 1.3: What are the KPIs of mobile application testing automation frameworks?

RQ 1.4: What are the challenges of integrating mobile application testing automation frameworks in Agile/DevOps environment?

RQ 2.1: How do the frameworks compare with each other with respect to the KPIs in Agile/DevOps environment ?

RQ 2.2: What are the capabilities of one popular framework compared with another?

3.2.1 Research Questions Part One

Part one of the research questions were discussed in Chapter 2. The main goal of this part of the project was to get an idea of the overall test automation frameworks and what metrics can be used to compare them.

Research question 1.1 deals with the identification of mobile application testing automation frameworks available in the market today. To answer this question, various blogs and research articles were examined. There is a plethora of frameworks available today and many articles were found talking about their usage and their capabilities. The research was narrowed down to Android testing automation frameworks which are open-source. The common frameworks found were - Espresso, Appium, Robotium, Calabash and MonkeyRunner.

Research question 1.2 deals with the ranking of the identified frameworks according to their popularity. As to the knowledge of the author, there are currently no scientific articles that discusses the popularity of the frameworks and therefore this research question was answered though informal means. Google Trends website and Stack overflow which is a website for programming related questions were used. By using two different sources, the results could be used if they are similar.

Research question 1.3 deals with identification of the KPIs to be used for comparing the different frameworks. To answer this, first various scientific papers that discuss comparison of frameworks were looked upon to observe the metrics which were used for evaluating. It was found that execution time was used in almost all the papers. The papers only mentioning test automation frameworks were also looked upon to get an idea of any metric that could be used such as flakiness.

Research question 1.4 deals with challenges of integrating the frameworks to the Agile/DevOps environment. Various papers and articles were looked to research upon the difficulties of implementing the CI and CD principles.

3.2.2 Research Questions Part Two

Part two of the research questions consists of collecting data and then obtaining results based on the empirical analysis to compare the different frameworks.

Research question 2.1 deals with comparing the frameworks based on the results obtained after performing analysis on the different KPIs that were identified. Many test cases were executed for the collection of data and then analysed. The results are presented in chapter 5.1 and then discussed in chapter 6.1.

Research question 2.2 deals with comparing the different capabilities among the different frameworks. The capabilities were identified through the execution of different test cases and observing what functionalities certain frameworks were capable of executing. A table showing the different capabilities is presented in chapter 5.2 and then discussed in chapter 6.2.

3.3 Research Paradigm

3.3.1 Measurement of the KPIs

1. **Execution time** – In order to measure the execution time for all the frameworks, the time taken for each test to execute without failure was measured. Each test's execution time was measured by running the test suite with the help of the pipeline at different times.

2. **Maintainability** – In order to calculate this metric, firstly the LOC for each test was measured. This was done taking into account that all the frameworks are trying to achieve the same functionality. Next, to check the efficiency of this metric in an Agile/DevOps environment, the changes in LOC required for all the frameworks to make the test pass after a particular release was also measured. This happens due to the fragility which is concerned with the frequent changes in the GUI of the application [25]. This can be measured as the number of LOC added, deleted or updated at a particular release for a test.

3. **Flakiness** – To measure flakiness, the number of waits required for each test was taken into account. The number of waits were measured with LOC that contain some wait statements. Waits were inserted in different places in the code in order to avoid the test failing due to the framework's inability of detecting the element quickly. Also, a flaky test was written in all the frameworks and the framework's capability to detect the element as soon as it appears was measured.

4. **Reliability** – This metric is measured as the number of tests that fail when they should not. There were no false negatives, that is the tests that pass when they should not and hence, was not considered in this study. A comparison can be performed by taking a ratio of the total number of times a test failed to the total number of times the test has been executed in each framework.

5. **Fragmentation** – Different devices may affect the framework's capability of detecting elements and this can have an impact on the result of the test case. The number of 'sleep' statements can be measured among different devices for different frameworks. 'Sleep' statements are responsible for making the current thread of the program sleep for specific duration. The reason 'sleep' statement was taken for measurement is that since measurement of fragmentation is basically the changes to be made in different devices, the devices required different number of 'sleep' statements due to the strange behaviour of the framework in certain devices. Also, some frameworks may require an alternative way of performing an action in certain devices and this can be measured as the number of alternate executions in different devices to make the tests pass.

6. **Identification of defects** – In some cases, it might be possible for the framework to identify a defect during the execution of the test cases. For comparison between frameworks, this can be measured as the number of defects found in the tests by the framework. Defect here refers to the test failing due to a bug in found in the mobile application which was probably not identified during the process of manual testing.

7. **Battery Usage** – This metric is measured by calculating the battery drain after execution of all the test cases for each framework. In order to obtain a fair result, the battery status is reset before every execution of test cases for each framework.

3.3.2 Evaluating the capabilities

In order to estimate the capabilities, test cases were implemented using the frameworks and then the difference in ease of executing various functionalities were observed and noted. These notes will give a rough estimate of what features the respective frameworks are capable of performing while executing several functionalities needed for automating the tests. This work will then give the reader a basis to decide which framework to use according to the requirements and features of the mobile application used for testing.

3.3 Data Collection

For the purpose of data collection, first a suitable mobile application was selected which had many flows and different UI elements that would give a representative performance measurement. Then, a set of tests were chosen in the selected mobile application to gather performance metrics. The tests were then implemented in all the frameworks using a similar code style. Next, a CI/CD pipeline was built with the help of Jenkins and the frameworks were integrated to the pipeline. The pipeline was then made to run at specified intervals and different performance metrics such as execution time were gathered and stored in the cloud. The tests were made to run in two different mobile devices in parallel to collect data from both the devices.

3.4 Experimental Design

3.4.1 General Agile/DevOps Environment setup tools

There are several stages that occur in an Agile/DevOps environment. Firstly, the developers push the code into a source repository. A VCS is used for this purpose. A VCS is mainly designed for pushing the code and tracking the iterative changes made in the code [33], which is imperative in an Agile environment where new code is pushed almost every hour. Git was used as the VCS and Bitbucket was used as the repository tool for this thesis. Then a build tool was chosen based on the type of project being implemented. For developing the test suites in Espresso and Robotium, Gradle was used as a build tool and for Appium, Maven was used. The next stage was choosing a CI server and integrating the VCS to it to build the code based on the selected build tool. For this study, Jenkins was used as the CI server. The next stage was testing according to the unit or integration test cases written by the developer. Jenkins runs test cases using tools such as Junit for unit testing. The final stage was the CD server for deploying the code into production and Jenkins was used as the CD server.

3.4.2 CI/CD Pipeline setup tools

For designing a pipeline, selecting the CI tool is an important decision. There are several CI tools available in the market today. In [34], Cruise Control, Hudson, Apache's Continuum, Jenkins and Team City were the CI tools that were compared. Among the different tools mentioned, Jenkins provides clearer dashboards, has plugins that can be easily added, can find bugs ,etc. Also, the build flow in Jenkins can be customized as the user wants, which cannot be done in the other tools easily. Moreover, Jenkins can also be used as a CD tool thereby making it possible to create a CI/CD pipeline using one tool itself. Groovy script was used to code the pipeline.

3.4.3 Test Environment setup

First a system was setup where the Jenkins master could run and this is called the master node. The Jenkins master was setup to build an APK every time a new code change has been merged into the repository of the mobile application(Bitbucket). Then another system was setup which acts as a slave to the Jenkins master and is called the slave node. Then a CI/CD pipeline script was written on the Jenkins server of the master node . The pipeline was scheduled at different intervals during the day and was made to run on the slave node. The slave node had two mobile devices connected where the latest version of the mobile app was installed and the tests would be run. The two different devices where connected to the slave node using USB cables.

The Espresso and Robotium test suites were stored in the same repository as the code of the mobile application(Bitbucket) and for Appium, a separate repository called Gerrit was used. All the frameworks were run on the pipeline built.

For building the APK, the command used was:

gradlew clean :<app package name> :<test package name>

For running the Espresso and Robotium test suite, the command used was:

adb shell am instrument -w -r <test package name> androix.test.runner.AndroidJUnitRunner

For running the Appium test suite, the command used was:

mvn -B -Dmaven.test.ignore verify

3.4.4 Hardware/Software used

Table 3.1: Hardware/Software Versions

Master Node OS	Linux Version 4.9.0-8-amd64
Slave Node OS	macOS Catalina Version 10.15
Mobile Device 1	Samsung Galaxy S9+ Android Version- 9
Mobile Device 2	Honor 5X Android Version – 5.1.1
Gradle Version	3.6.0
Maven Version	3.6.3
Espresso Version	3.2.0
Robotium Version	5.6.3
Appium Version	7.3.0
Jenkins version	2.235

For the purpose of this study, two mobile devices - Samsung and Honor were chosen as the former is a fast device and the latter is a slow device which would give us good estimation about different KPIs in different devices.

3.5 Assessing Reliability and Validity of Data Collected

3.5.1 Reliability

The literature survey was based on legitimate information sources such as Google Scholar, IEEE and Springer Link. To ensure that the results collected were reliable, the mobile devices where the tests have been run were rebooted before every run of the test to reduce the background services running. The mobile application was uninstalled and installed again before every run. Also, the same version of the mobile application was used for running the tests in all the frameworks in a single run.

3.5.2 Validity

To check the validity of the collected results, the data collected in a particular run for a specific test was compared with the data from previous runs of the same test to make sure that there was not a large difference in terms of values unless some change has been implemented in the test. We ensured that no other applications were running on the mobile device during the execution of the tests that might have affected the data collected.

3.6 Planned Data Analysis

For analysing the data collected, RStudio was used. Statistical analysis was done on the multiple execution times obtained by the different frameworks using the two sample t-test. This gave an estimate on the significance of the execution time in the frameworks. For other KPI metrics, quantitative analysis and exploratory data analysis were done based on the data collected.

4. Methods

This chapter gives the details of how the experiments were conducted to gather results and meet the goals of this thesis.

4.1 Selection of Mobile Application

In order to perform a suitable comparison of mobile application testing automation frameworks that can be used by the industry, a mobile application that is currently active in the market and used by millions of users across the world needed to be chosen.

Truecaller is a mobile application having the main functionality to identify the caller when you receive a call from an unknown user. The application provides an all-in one app where the user can call, message and search mobile numbers within the application itself. The application also provides advanced blocking options to minimise the effect of spammers as much as possible.

The application is ideal for evaluating the performance of frameworks with respect to UI testing as the application has a variety of different flows with different UI elements that can be used to benchmark the performance of a framework. Moreover, the application also has a variety of different functions that can be used for functional testing and evaluate the performance of the framework.

Truecaller has a very strong user base and a very good rating. Moreover, the application also has frequent updates almost every hour during development thus making it an ideal candidate to use in an Agile/DevOps environment. Selecting a real-world application will also make it possible for the tests to be developed similarly as in an industry setting.

4.2 Selection of test cases

The selection of test cases is an important issue as they should be selected in such a way that measurements for the different KPIs of mobile application testing automation frameworks could be obtained. Each of the test cases has a set of steps followed. Most of them were implemented to execute some functionality of the mobile application. The test cases were selected based on the guidelines from the App Quality Alliance testing guide [35] and those which give a good performance measurement.

AQuA is a global association that focuses on helping the industry continually improve and promote mobile app quality. It is responsible for producing testing criteria for different platforms against which developers can test their apps. The list from AQuA had to be modified for automatic GUI testing as it included testing in a more general form and there were some parts such as interacting with hardware buttons that were not relevant for the scope of this study. AQuA was chosen as it gives some guidance on what should be tested. Table 4.1 shows the AQuA tests chosen.

Table 4.1: AQuA tests selected

Name of Test	**Description of Test**
Functionality Check	Test the functionality of the application to make sure it does what the function is supposed to do (simplified version from AQuA)
Scrolling in menus	Scrolling in navigation drawer and other scrollable lists

Table 4.2 shows the test cases that have been selected and implemented for all the mobile application testing automation frameworks in the selected mobile application. The detailed steps of each test are given in Appendix A.

Table 4.2: Test cases implemented on mobile application

Name of Test Case	Description	Selection Criteria	AQuA test
T1	Registration of the user	Consists of many flows through different screens which can be used to measure battery consumption and performance measurement	Functionality Check
T2	Creating Business Profile	Consists of inputting several different types of details which could test the performance of the framework	Functionality Check & Scrolling in menus
T3	Counting the toggles checked, updating one toggle and checking the count of all toggles	Gives a good performance measurement of the scrolling behaviour due to selection of last toggle of the screen as well as checking state of all toggles in that screen	Functionality Check & Scrolling in menus
T4	Sending chat message	Involves reading different types of elements for a good performance measurement	Functionality Check
T5	Initiating VoIP Call	Involves reading different types of elements for a good performance measurement	Functionality Check
T6	Block and Unblock Contact	Involves identification of several UI elements which can give a good performance measurement	Functionality Check
T7	Create, Edit and Leave Group Chat	Involves multiple flows which can give a good performance measurement	Functionality Check
T8	Check the count of unread messages	Involves several flows and performing different types of operations such as long click that can measure the performance of the framework	Functionality Check
T9	Block, Check Block List, Unblock Contact	Involves several flows and identification of different elements	Functionality Check & Scrolling in menus
T10	Swipe Through Different Descriptions	Demonstrates the performance of different frameworks with respect to swiping the screens and detecting some element quickly in the new swiped screen	Functionality Check
T11	Block Number Using Floating Action Button Menu Items	Consists of using different elements such as Floating Action Button Menu items that can give a good performance measurement	Functionality Check & Scrolling in menus

T12	Change Toggle State, Deactivate, Check Toggle State	Involves several flows that can test the battery consumption and performance of the framework	Functionality Check
T13	Check Element for Flakiness	Introduced for the purpose of checking the degree of the frameworks to handle flaky tests involving navigating to a particular page where an element will appear but the time it takes to appear will vary since the element is fetched from a backend resource	Functionality Check
T14	Save and Remove Contact	Demonstrates the selection of elements outside the AUT capability in Appium. It consists of going to the native contacts app present in the device from AUT and saving a new contact and then removing the contact from the AUT	Functionality Check
T15	Verifying Call from Blocked Contact End to End	Demonstrates the end-to-end test capability of multiple devices in Appium where certain portions of the same test are run on different devices. This test consists of blocking a contact from one device, and then calling from another device who has been blocked and then verifying whether a blocked call message is present in the spam tab	Functionality Check

4.3 Implementation of KPI measurements

1. Execution Time

For Appium, 'extent reports' was used as the reporting framework and TestNG was used as the testing framework to monitor test executions. Extent reports is an open-source library that generates customizable HTML reports [36]. TestNG is a testing framework with several functionalities such as listeners. For the purpose of this project, TestNG listeners were used and integrated with extent reports to generate the execution time for each test case.

For Espresso and Robotium, no ideal reporting framework was found that could monitor execution time of each test individually rather than the total time of all the tests. Therefore, for the purpose of this project, the time at the start of each test was measured and then the time at the end of the same test was taken and the difference was calculated to be the execution time for that test.

For the sake of validity, the author checked the execution time used in the above method with the framework Appium and it was found that the reporting framework (Extent reports) returned the exactly the same values as from the methods used in Espresso and Robotium.

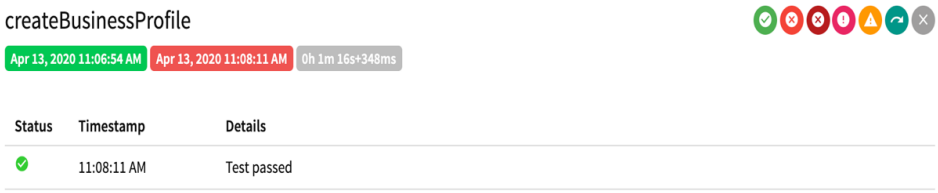


Figure 4.1: Extent Report of Passed Test

Figure 4.2: Extent Report of Failed Test

2. Maintainability

The measurement of LOC was done using the tool called CLOC and also manually. CLOC allows the user to specify the name of the file to count the number of lines while ignoring blank lines. The use of import statements was not taken into consideration for the measurement.

For the sake of validity, the code style used for all the frameworks was the same and written solely by the author. The figure below shows the output from running the CLOC tool.

```
      1 text file.
      1 unique file.
      0 files ignored.

github.com/AlDanial/cloc v 1.84  T=0.02 s (45.3 files/s, 28456.2 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                             1             62            113            453
-------------------------------------------------------------------------------
```

Figure 4.3: CLOC tool output example

3. Flakiness

The LOC of the wait statements were done manually by the author. The number of failures for the flaky test in each framework were also done manually by observing whether the test passed or failed.

4. Reliability

For Appium, the Extent report dashboard gave a clear representation of the number of tests that failed and passed for each execution. For Espresso and Robotium, a counting variable was defined in the code which was incremented every time a test has failed during execution.

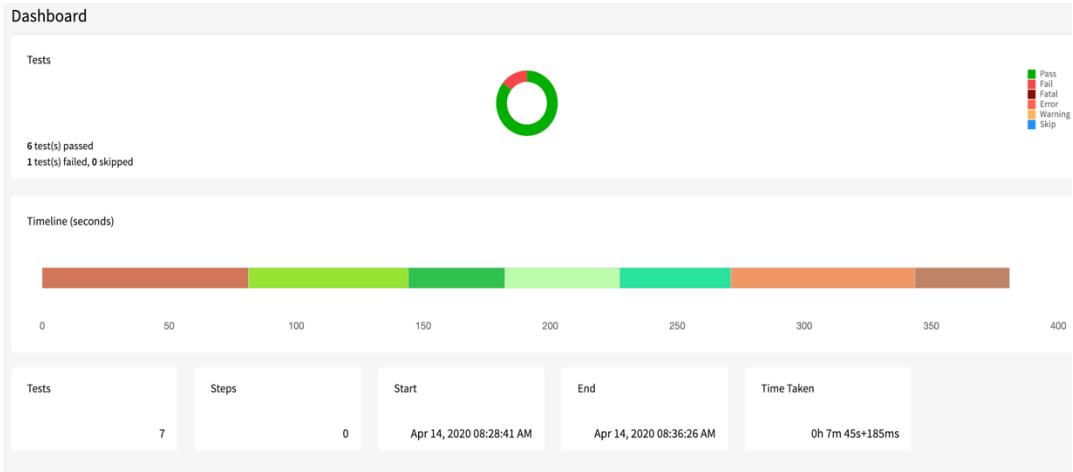


Figure 4.4: Extent Report Dashboard

5. Fragmentation

The measurements were done manually by the author by counting the number of sleep statements and alternate executions throughout the codebase of each framework.

6. Identification of defects

The measurements were done manually by the author by observing the tests that failed due to defect in the application and the reason why they failed.

7. Battery Usage

In order to measure the battery statistics, "adb" command is used.

The full command used was:

adb shell dumpsys batterystats –charged <app package name>

Through the usage of this command, it generated aggregated observations about battery use in the device since it was last charged [37]. This command gives a large amount of device information and it was filtered to get only obtain the battery usage. The output is measured in terms of mAh.

For the measurement, it was imperative that the battery statistics of the devices are reset before each measurement in order to get accurate results. Since the command gives information since the last time the battery has been charged, it is necessary to unplug the device before every execution.

The command used to unplug the device was:

adb shell dumpsys battery set status 3

This command does not necessarily disconnect the device from the system but sets the battery status to discharging. Finally once the test execution is done, the battery status can be set to charging again.

The command used to set the battery status to charging again was:

adb shell dumpsys battery reset

4.4 Implementation of the CI/CD pipeline

The design of the pipeline consisted of different pipeline stages that are responsible for particular functions. Each stage executes only after the previous stage has finished execution. All the commands to be executed on the command line are run in the Jenkins pipeline with the help of Shell Script commands. The pipeline was built using a declarative syntax. The different stages are –

1. **Stage(Prepare build directories) –** This stage is responsible for creating the directories on the system where the respective code will be pushed to.
   The directories are created using the command:

   mkdir <directory name>

   The directories are created to store the Appium Test suites and Espresso and Robotium Tests after fetching them from the corresponding repository.

2. **Stage(Clone and Build resources) –** This stage is responsible for fetching the code from the corresponding git repositories and also building an APK of the mobile app. In order to fetch from the repository, the command "git" is used along with passing the credentials and the URL and the branch as parameters.

For building the APK, the command used is:

gradlew clean :<app package name> :<test package name>

The APK is built for the latest version that has been pushed into the code repository at the time the pipeline is run. This version keeps updating during every run of the pipeline depending on the code pushed into the repository. This implements the CI principle where the latest version of the code is automatically fetched from the repository at the time when the pipeline is scheduled to run.

3. **Stage(Run Espresso Test Suite) –** This stage consists of running a python script which is responsible for rebooting the device, installing the application and running the tests for Espresso test suite multiple times until all the tests have passed once or the tests have failed two times. Finally , the script also fetches the battery information from the device and prints in the console. This stage is performed concurrently in both the connected mobile devices. The pipeline is made to continue even though the stage might fail to finish on outcome of failure of certain tests so that the next stage of the pipeline can be executed.
4. **Stage(Run Robotium Test Suite) –** This stage consists of running the same python script but this time the test suites for Robotium are run. The python script can distinguish between the different frameworks with the help of an argument passed. This stage is also done concurrently for both the connected mobile devices and like the previous stage, the pipeline is made to continue even though this stage might fail.
5. **Stage(Run Appium Test Suite) –** This stage consists of running the same python script but this time the test suites for Appium are run. This stage is also done concurrently for both the connected mobile devices and the pipeline will continue irrespective of the result of this stage.
6. **Stage(Run Verifying Call from Blocked Contact End to End test) –** This stage consists of running the 'Verifying Call from Blocked Contact end to end' test only in the framework Appium. This is done directly without any python script by running the maven command on the console and passing the devices as arguments.

After all the stages are completed, there is a "post" stage which is responsible for displaying the results of all the previous stages in terms of success or failure and finally there is an "always" stage which gets executed regardless of success or failure and is responsible for resetting the battery status of the device and shutting it down. The built pipeline was made to run at specific schedules in the slave node where the mobile devices were connected and the data collected was stored into the cloud for analysis during a later stage.

To implement the CD principle, if all the tests have passed, then Jenkins will initiate the release process of the latest version of the app. In an ideal scenario, the app should be released into production. However, due to the large number of functionalities present in the mobile app and short span of time available, it was not possible to automate all the tests. Thus for this project, the release of the latest version of the app was limited to an alpha version. Moreover, since not all the test cases could be implemented within the time span available, the continuous delivery of manually releasing the app into the alpha version was implemented since some test cases had to be tested manually rather than the continuous deployment principle where the release process happens automatically.

4.4.1 Overcoming challenges of integration of mobile application testing automation frameworks into the pipeline -

1. **Unstable Test cases–** If there are frequent failures of tests, this will result in the pipeline failing and this could result in the lack of a successful build. This will affect the release process as the latest version of the mobile app will never be released. In order to overcome this, the author has selected stable test cases. Such test cases should not fail often other than from problems with the mobile application or the testing framework to reduce uncertainty [38]. Also, in case there is a failure during an execution of the tests, all the tests are re-executed instead of directly making the pipeline fail. This reduces the probability of the pipeline failing due to flaky tests i.e. tests which fail during some runs but not always. However, if the tests fail again after re-execution, then it is most likely the fault of the framework. It is to be noted that in an ideal scenario, only the tests that failed would be re-

executed instead of all the tests. But in this study, since certain measurements of KPIs required all the tests to be executed together, all the tests were re-executed in case of failure of even a single test.

2. **Lack of Proper Infrastructure –** To overcome the problem of having an infrastructure where the integration of mobile application testing automation frameworks becomes very challenging, we chose to implement the pipeline on Jenkins which provides a wide range of functionalities and security features. Through the division of stages in pipelines, it was possible to control where the integration of test automation frameworks should take place. The master node was made to connect to the slave node through the SSH protocol. SSH is a protocol for secure remote login over an insecure network [39]. To fetch the code from the repositories, the groovy script on Jenkins uses an additional parameter called authentication present with the git command to allow only authorized access of the code repositories.
3. **Distributed Organization –** This problem of test code duplication and merging issues did not happen in this project as all the code was developed solely by the author. But the current setup can be used in future as well as the code can be merged only after it has been reviewed and approved by someone other than the person who has written the code with the help of rules enforced on the CI pipeline. Also, only one pipeline has been implemented on one CI server for running automation tests, avoiding any merging issues that might occur due to multiple CI servers running the same automation tests. Moreover, a notification is sent to all the concerned parties after a single run of the pipeline thus keeping everyone aware about the current state of the pipeline and any issues that need to be addressed.
4. **Build Time –** In order to execute the tests concurrently in two different devices at the same time, Jenkins provides a "parallel" construct within a single pipeline stage. This can reduce the build time enormously as the tests can be executed at the same time to collect data from two different devices rather than wait until one test completes before starting the same test in a different device. It is also possible to split the independent tests in two different mobile devices to reduce the build time further, thereby also reducing the feedback time from the execution of the test cases.
   The author also attempted to implement the CRTS and CTSP techniques as described in [27]. The main motivation behind these techniques is to select which tests to execute to avoid excessive resource consumption at the pipeline. The authors in [27] have described a set of parameters to achieve test selection and test prioritization in terms of W_f as a failure window, W_e as an execution window and W_p as a prioritization window, where each window can be defined as a number of test suites. For each window, the authors have considered the tests that fail within W_f, the tests that have been executed within W_e and the tests that are going to be executed within W_p. However, in this project, since three frameworks were used, the notion of tests failing in W_f can only be applicable if the same test was failing in all the frameworks at the same time. Since there were a negligible number of tests that fail in all the frameworks, it was not possible to select an appropriate window size for W_f. Moreover, since the main goal of this study was to compare all the tests implemented to get maximum performance comparison, there would be no notion of windows W_e and W_p and hence the techniques were excluded from our study although they could be used in the pipeline if there was only one framework and there were more tests implemented [40].
5. **Awareness and Visibility of Build and Test Results –** For an easy and quick interpretation of results of the pipeline, an additional plugin was added to Jenkins called Blue Ocean [41] . It provides a user interface through which it is possible to see the result of different stages of the pipeline in a graphical view along with the detailed description on clicking the corresponding stage. Thus, reading unnecessary information can be avoided and also can result in faster feedback of the build and test results.
   The author has found that the interpretation of results through Blue Ocean corresponds to partial use of the CIViT technique mentioned in [32]. CIViT is concerned with three different aspects - types of testing, scope of testing and periodicity of testing. Among the different types of testing supported by CIViT, only the new functionality type of testing was needed for the scope of this thesis. New functionality testing is the testing of the functionality that is currently under development [32]. Among the different scopes of testing which refer to the segment of the overall system tested, subsystem testing was used as the scope due to the wide range of functionality tested among different teams. Finally with regards to periodicity of testing which corresponds to the time between the start of the

testing activity and the availability of feedback from the testing activity, the level used was minutes and hours due to the time taken for all the pipeline stages to finish execution.

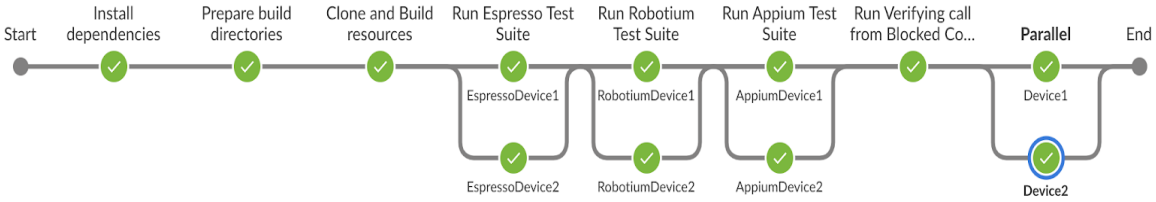


Figure 4.5: Blue Ocean view of the Pipeline

From Figure 4.1, the different stages of the pipeline can be observed. The stages 'Run Espresso Test Suite', 'Run Robotium Test Suite', 'Run Appium Test Suite' and 'Parallel' were run concurrently in the two connected mobile devices. The last 'Parallel' Stage is the 'always' stage for resetting the battery stats of the two devices.

5. Results and Analysis

This section describes the results and the analysis performed and more detailed discussion is given in Chapter 6.

5.1 Measurement of KPIs

The empirical analysis was performed on the KPIs for all the frameworks based on the data collected. The extra two tests implemented only in Appium to indicate the framework's capability were not included in the empirical analysis.

5.1.1 Execution Time

The execution time (in seconds) for all the test cases in the Samsung device are presented in Table 5.1.

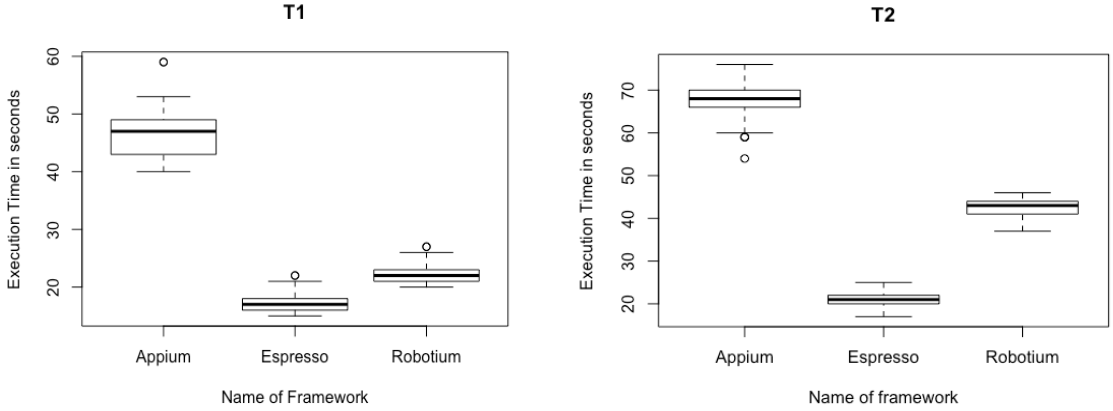
Table 5.1: Execution time of different test cases in Samsung device

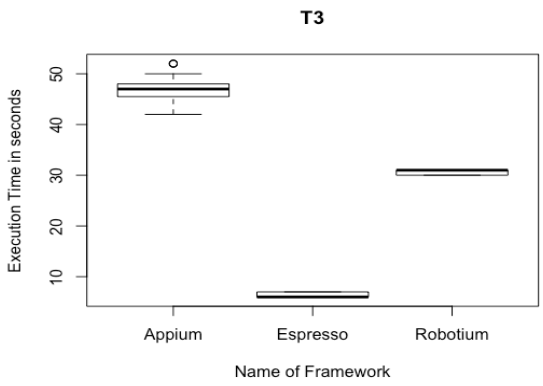
Test Case		**Appium**		**Espresso**		**Robotium**	
T1	Mean	46.681		17.361		22.625	
	Std. Deviation	4.114		1.841		2.072	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T2	Mean	68.197		21.057		42.437	
	Std. Deviation	4.566		1.676		1.849	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T3	Mean	46.979		6.417		30.729	
	Std. Deviation	2.274		0.498		0.449	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T4	Mean	37		7.181		14.625	
	Std. Deviation	1.610		0.969		0.680	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T5	Mean	38.111		16.958		24.597	
	Std. Deviation	2.43		1.326		1.109	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T6	Mean	41.611		16.319		30.681	
	Std. Deviation	3.462		1.471		1.111	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T7	Mean	28.063		9.601		33.478	
	Std. Deviation	1.21		0.537		0.505	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T8	Mean	70.694		30.167		68.667	
	Std. Deviation	4.130		1.583		1.792	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000

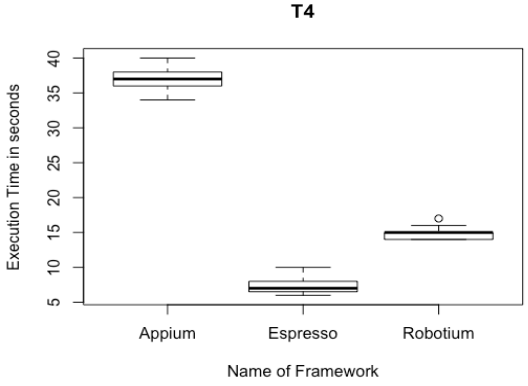
T9	Mean	46.333		24.167		39.167	
	Std. Deviation	2.266		0.519		0.663	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T10	Mean	55.438		12.625		41.435	
	Std. Deviation	2.324		0.606		0.501	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T11	Mean	34.458		13.417		41.257	
	Std. Deviation	2.583		0.868		3.291	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T12	Mean	87.606		32.861		71.569	
	Std. Deviation	2.94		1.871		2.213	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000

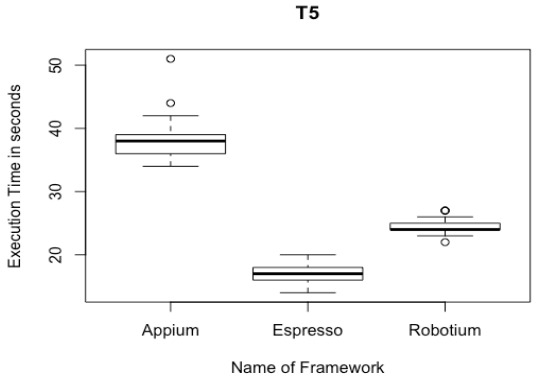
The Welch Two Sample t-test was conducted for all the test cases. This can be used to test whether the two samples have equal means or not. A null hypothesis can be assumed with the meaning that the means of the two different samples are equal. The p-value also known as the probability value is the probability of obtaining the results as extreme as the observed results of a statistical hypothesis test, assuming that the null hypothesis is correct [42]. A smaller p-value would indicate that there is a stronger evidence in favour of the alternate hypothesis indicating that the means of two samples are not equal. A commonly used significance level is 0.05 and if the p-value is less than 0.05, then there is evidence against the null hypothesis.

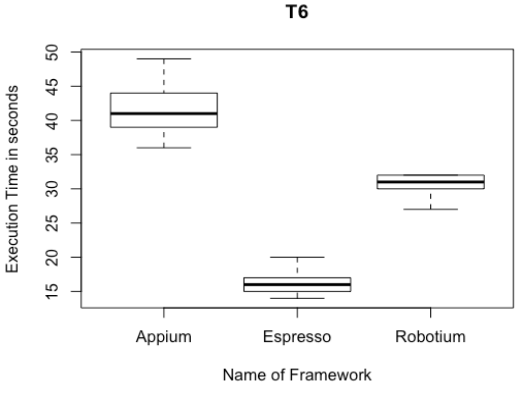
From Table 5.1, since the p-values for all the test cases were 0.000 which is lower than 0.05, it can be said that the differences between mean execution times of each framework are significant. The framework Espresso is the fastest for all the test cases among the three frameworks due to the quick detection of elements and execution of actions. There are only two test cases where Robotium is slower than Appium otherwise Appium is the framework having the longest execution time in the test cases. The boxplots for the results of execution time in the Samsung device are presented below.

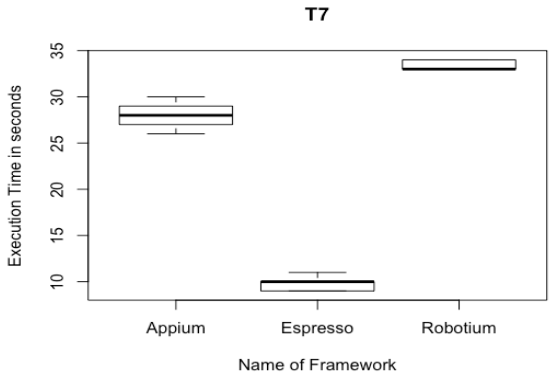


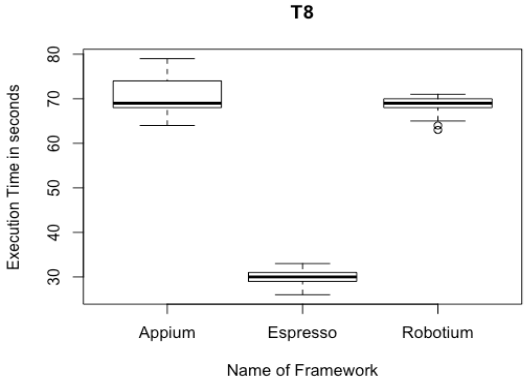

T3
Execution Time in seconds
10
20
30
40
50
Appium
Espresso
Robotium
Name of Framework

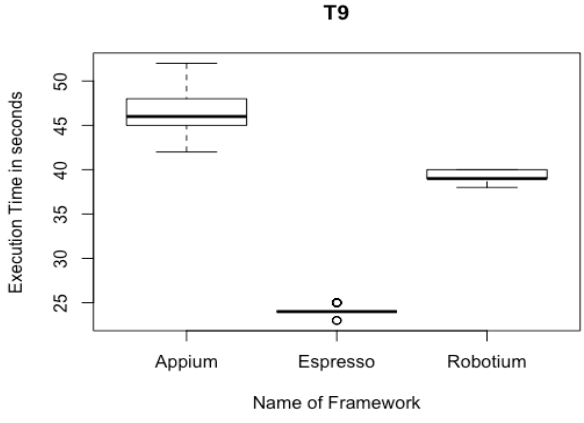

T4
Execution Time in seconds
5
10
15
20
25
30
35
40
Appium
Espresso
Robotium
Name of Framework

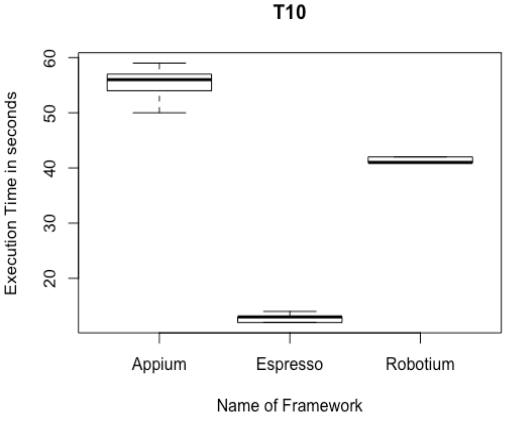

T5
Execution Time in seconds
20
30
40
50
Appium
Espresso
Robotium
Name of Framework


T6
Execution Time in seconds
15
20
25
30
35
40
45
50
Appium
Espresso
Robotium
Name of Framework


T7
Execution Time in seconds
10
15
20
25
30
35
Appium
Espresso
Robotium
Name of Framework


T8
Execution Time in seconds
30
40
50
60
70
80
Appium
Espresso
Robotium
Name of Framework


T9
Execution Time in seconds
25
30
35
40
45
50
Appium
Espresso
Robotium
Name of Framework


T10
Execution Time in seconds
20
30
40
50
60
Appium
Espresso
Robotium
Name of Framework

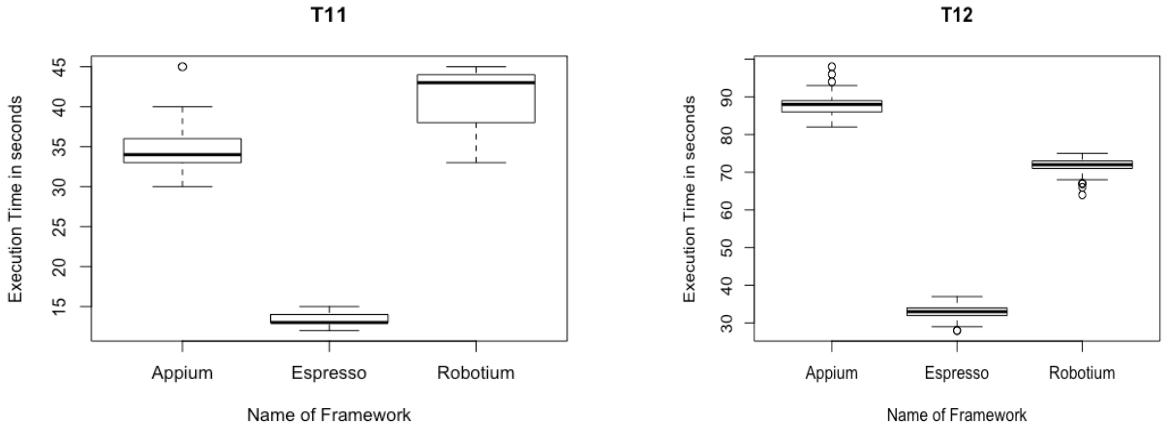


Figure 5.1: Boxplots of results of execution time in Samsung device

From Figure 5.1, it can be observed that among all the frameworks, Appium has the largest variation in execution times in almost all the test cases except T11 which can also be confirmed by looking at the standard deviation values from Table 5.1. It can also be noticed that all the frameworks have some outliers as well (through the indication of circles in the above Figure) with Appium having them in the maximum number of test cases.

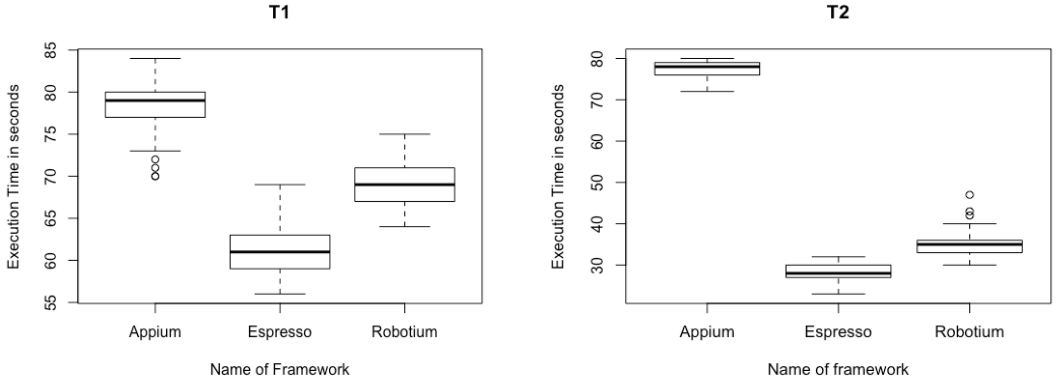
Table 5.2: Execution time of different test cases in Honor device

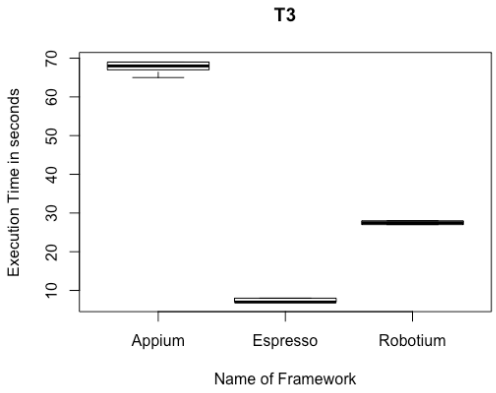
Test Case		**Appium**		**Espresso**		**Robotium**	
T1	Mean	78.361		61.208		69.181	
	Std. Deviation	3.046		2.545		2.468	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T2	Mean	77.577		28.203		35.114	
	Std. Deviation	2.109		2.097		2.922	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T3	Mean	67.854		7.396		27.5	
	Std. Deviation	0.989		0.494		0.505	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T4	Mean	40.069		12.486		15.389	
	Std. Deviation	1.763		0.934		1.217	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T5	Mean	45.93		21.014		25.786	
	Std. Deviation	1.751		1.25		1.579	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T6	Mean	43.528		20.375		25.986	
	Std. Deviation	1.50		2.052		0.942	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000

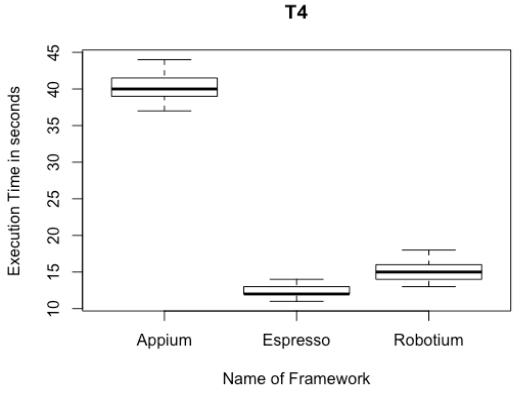
T7	Mean	32.383		13.444		27.543	
	Std. Deviation	1.328		0.503		0.504	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T8	Mean	74.306		36.639		57.465	
	Std. Deviation	2.751		1.568		1.026	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T9	Mean	43.067		31.667		35	
	Std. Deviation	0.72		0.781		0.885	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T10	Mean	52.125		15.917		34.543	
	Std. Deviation	0.761		0.794		0.959	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T11	Mean	38.486		18.708		32.042	
	Std. Deviation	1.363		0.895		0.659	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000
T12	Mean	98.286		54.764		81.186	
	Std. Deviation	2.121		2.755		1.158	
	P-value	Espresso	0.000	Appium	0.000	Appium	0.000
		Robotium	0.000	Robotium	0.000	Espresso	0.000

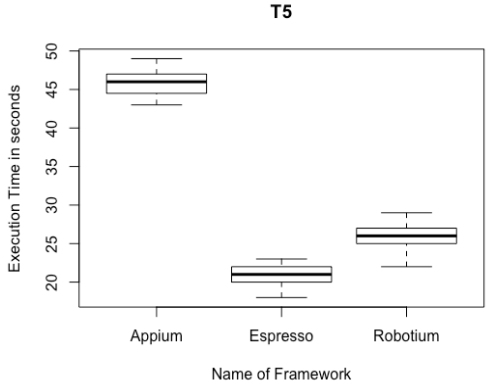
The Welch Two Sample t-test was again conducted for all the test cases and the p-values for all the test cases were again found to be 0.000 which is lower than 0.05 and hence it can be said that the differences between mean execution times of each framework are significant in Honor device as well.

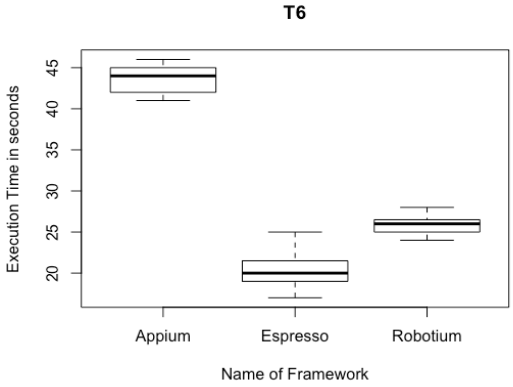
From Table 5.2, it can be observed that in the Honor device, the framework Espresso is again the fastest for all the test cases while the framework Appium was found to be the slowest in all the test cases and the level of speed again varies depending on the test case. On comparing the execution times of the frameworks in the Honor device with the Samsung device, the frameworks have a longer execution time in the Honor device. The boxplots from the execution time in the Honor device are presented below.

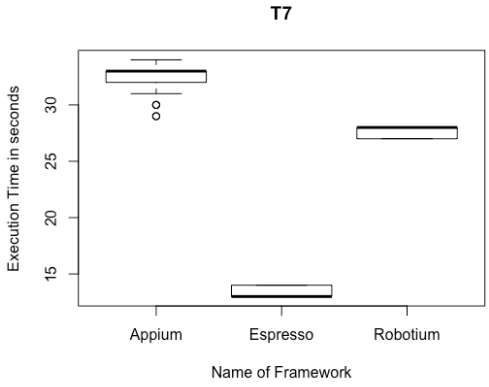


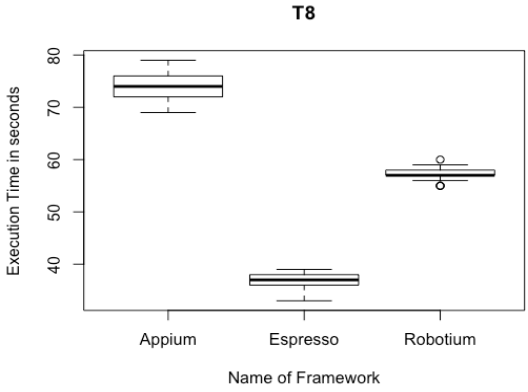

T3
Execution Time in seconds
10
20
30
40
50
60
70
Appium
Espresso
Robotium
Name of Framework

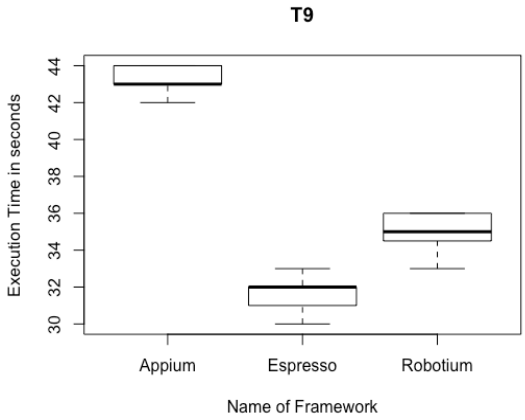

T4
Execution Time in seconds
10
15
20
25
30
35
40
45
Appium
Espresso
Robotium
Name of Framework

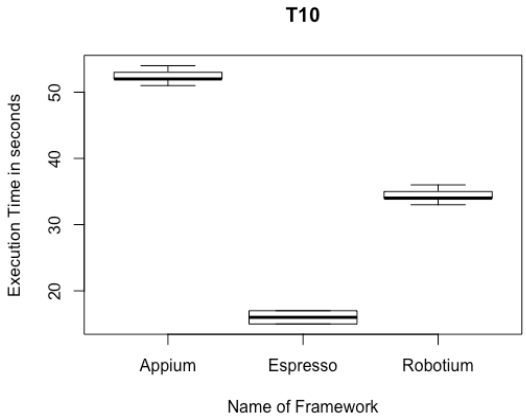

T5
Execution Time in seconds
20
25
30
35
40
45
50
Appium
Espresso
Robotium
Name of Framework


T6
Execution Time in seconds
20
25
30
35
40
45
Appium
Espresso
Robotium
Name of Framework


T7
Execution Time in seconds
15
20
25
30
Appium
Espresso
Robotium
Name of Framework


T8
Execution Time in seconds
40
50
60
70
80
Appium
Espresso
Robotium
Name of Framework


T9
Execution Time in seconds
30
32
34
36
38
40
42
44
Appium
Espresso
Robotium
Name of Framework


T10
Execution Time in seconds
20
30
40
50
Appium
Espresso
Robotium
Name of Framework

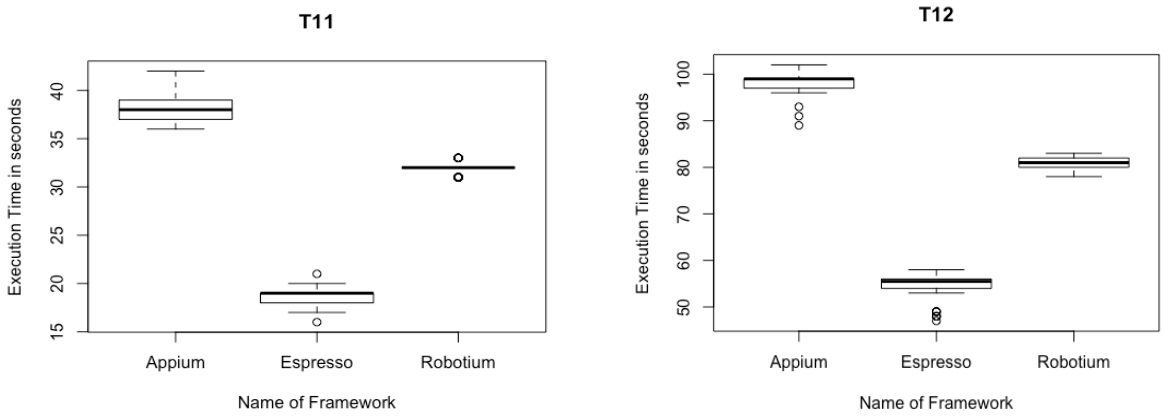


Figure 5.2: Boxplots of results of execution time in Honor device

From Figure 5.2, it can be observed that Appium again has the largest variation in execution time compared to the other frameworks and all the frameworks have some outliers which are overall more compared to the count in Samsung device.

5.1.2 Maintainability

The Table 5.3 summarizes the total LOC added, modified or deleted in different releases per test case in different frameworks. Only the test cases that required change in the code were taken into consideration for this measurement.

Table 5.3: Change in LOC per framework

Test Case	**Appium**	**Espresso**	**Robotium**
T1	14	12	10
T2	53	40	45
T3	8	2	2
T4	1	1	1
T7	2	2	6
Total	**78**	**57**	**64**

For the same change in functionality, Appium requires more changes in LOC while Espresso requires the least.

The total LOC for execution of the test cases in different frameworks is given in Table 5.4. Since all the frameworks are trying to achieve the same functionality, it will be interesting to see which framework requires less LOC to perform the same functionalities. The LOC was calculated from the test cases written solely by the author and following a similar code style. All the import statements were excluded

from the measurement. The LOC for setting up the framework was also calculated. The framework having the least LOC would be the best for maintainability.

Table 5.4: Total LOC per framework

LOC	**Appium**	**Espresso**	**Robotium**
From Test Cases	389	368	371
From setting up the framework	20	6	8
Total	**409**	**374**	**379**

The measurement was done using the CLOC tool by passing the path of the code file. Then the import statements from all the files were subtracted manually. Since all the three frameworks were written in the same coding language (java), the comparison can be justifiable. The import statements were excluded as the imports are commonly done by the editor and would not play a major factor in the maintainability of the code. The author also made sure that no blank lines were taken into consideration through usage of the CLOC tool which excludes the blank lines in the measurement. The LOC from the tests- T14 and T15 were excluded as these two tests were only implemented in Appium to indicate the framework's capability. The LOC for creating screenshots when the tests fail were also excluded for the measurement.

From Table 5.4, it can be observed that Appium again has the largest LOC while Espresso has the least making it best for Maintainability.

5.1.3 Flakiness

The number of wait statements per framework is given in Table 5.5. Wait statements refer to those statements that have framework specific wait methods to wait for an element until it appears. Appium required more number of wait statements while Espresso required the least.

Table 5.5: Number of Wait statements per framework

Framework	**Number of Wait Statements**
Appium	144
Espresso	47
Robotium	102

To further test the degree of the frameworks to handle flaky tests, a flaky test (T13) was implemented which depends on the element being fetched from the network and hence the time for the element to get displayed may vary but, an upper bound can be estimated on the time before which the element is guaranteed to appear provided the same network is used. Based on the test, it can be detected how quickly the element can be identified by the frameworks once it appears after waiting for a period of time.

The test implemented was fetching an element from a backend resource. After executing the test cases manually for several times, it was found that the upper bound for the element to appear in the Samsung device is 9 seconds and the upper bound for the element to appear in the Honor device is 17 seconds. Thereby, the frameworks were set to wait until the element is detected for the time equivalent to the upper bound for the corresponding device and the test will fail if the framework fails to detect the element within that amount of time. Also, in all the frameworks, the author has implemented the screenshot of the current screen in the device when the tests fail. Through this, the author can manually

verify whether the test actually failed due to the framework's inability to detect the element or due to the element not being present.

Table 5.6: Number of failures divided by total number of test runs in T13 per framework

Framework	**Number of failures/ Total number of test runs**
Appium	8/112 (7.14 %)
Espresso	0/112 (0 %)
Robotium	4/112 (3.57 %)

From Table 5.6, it can be observed that Appium has the maximum number of failures with respect to the flaky test and Espresso has no failures at all.

5.1.4 Reliability

Table 5.7 shows the number of times the test failed divided by the total number of times the test has been executed in all the frameworks and the devices. The percentage of failure is indicated in the brackets.

Table 5.7: Number of failed tests divided by total number of tests per framework

Test Case	**Samsung Device**			**Honor Device**		
	Appium	**Espresso**	**Robotium**	**Appium**	**Espresso**	**Robotium**
T1	0/72 (0%)	0/72 (0%)	0/72 (0%)	0/72 (0%)	0/72 (0%)	0/72 (0%)
T2	1/72 (1.39%)	2/72 (2.78%)	1/72 (1.39%)	1/72 (1.39%)	3/72 (4.17%)	2/72 (2.78%)
T3	0/48 (0%)	0/48 (0%)	0/48 (0%)	0/48 (0%)	0/48 (0%)	0/48 (0%)
T4	0/72 (0%)	0/72 (0%)	0/72 (0%)	0/72 (0%)	0/72 (0%)	0/72 (0%)
T5	0/72 (0%)	0/72 (0%)	0/72 (0%)	1/72 (1.39%)	0/72 (0%)	2/72 (2.78%)
T6	0/72 (0%)	0/72 (0%)	0/72 (0%)	0/72 (0%)	0/72 (0%)	0/72 (0%)
T7	0/48 (0%)	2/48 (4.2%)	2/48 (4.2%)	1/48 (2.1%)	3/48 (6.25%)	2/48 (4.2%)
T8	0/72 (0%)	0/72 (0%)	0/72 (0%)	0/72 (0%)	0/72 (0%)	1/72 (1.39%)
T9	3/48 (6.25%)	0/48 (0%)	0/48 (0%)	3/48 (6.25%)	0/48 (0%)	1/48 (2.1%)
T10	0/48 (0%)	0/48 (0%)	2/48 (4.2%)	0/48 (0%)	0/48 (0%)	2/48 (4.2%)
T11	0/72 (0%)	0/72 (0%)	2/72 (2.78%)	0/72 (0%)	0/72 (0%)	0/72 (0%)
T12	1/72 (1.39%)	0/72 (0%)	0/72 (0%)	2/72 (2.78%)	0/72 (0%)	2/72 (2.78%)
Total	**5/768 (0.65%)**	**4/768 (0.52%)**	**7/768 (0.91%)**	**8/768 (1.04%)**	**6/768 (0.78%)**	**12/768 (1.56%)**

From the Table 5.7, it can be observed that all the frameworks have some number of failed tests in the Samsung device although theoretically, it should be zero. In Appium, the main test was failed frequently in the Samsung device was T9. Robotium had the maximum number of failures while Espresso had the least in the Samsung device. A similar trend can be observed in the Honor device as well with Robotium again having the maximum number of failures and Espresso having the least. Also, the number of failures in the slower Honor device is more than in the faster Samsung device for each framework. Espresso seems to be the most reliable framework while Robotium seems to be the most unreliable.

5.1.5 Fragmentation

One of the major code changes that could result in different devices is the number of 'sleep' statements which are basically the statements that make the current thread of program stop the execution for some amount of time before continuing. This is different from wait statements as the wait statements are framework specific which is associated with waiting only until some particular element appears but the 'sleep' statements are made to stop the execution for the amount of time specified. The 'sleep' could be needed in various scenarios during execution of certain functionalities where the framework specific wait method fails to detect the element in some devices. Another measurement of fragmentation is the number of alternate executions that need to be done depending on the device's inability to perform some action using the same code.

Table 5.8: Number of sleep statements and alternate executions in Espresso framework

Test Case	Sleep Statements	Alternate Executions
T2	2	1
Total	**2**	**1**

From Table 5.8, it can be observed that Espresso requires just two sleep statements and one alternate execution and both of them were from the same test case.

Table 5.9: Number of sleep statements and alternate executions in Robotium framework

Test Case	Sleep Statements	Alternate Executions
T1	2	-
T5	-	1
T7	-	1
T11	1	1
T13	2	-
Total	**5**	**3**

From Table 5.9, it can be observed that Robotium requires a large number of sleep statements and more alternate executions compared to Espresso.

Table 5.10: Number of sleep statements and alternate executions in Appium framework

Test Case	Sleep Statements	Alternate Executions
T1	2	-
T6	1	-
T11	-	1
T13	1	-
Total	**4**	**1**

From Table 5.10, Appium is observed to require four sleep statements and a single alternate execution making it better than Robotium but second best to Espresso.

5.1.6 Identification of defects

The identification of defects which were not found during the manual testing process by the frameworks plays a major role in the performance of each framework. There were only two defects which were observed as shown in Table 5.11 and all the frameworks were able to identify both of them.

Table 5.11: Identification of defects by the frameworks

Defect	Appium	Espresso	Robotium
Contact not visible	Yes	Yes	Yes
VoIP Icon not visible	Yes	Yes	Yes

5.1.7 Battery Usage

For measurement, it is imperative that the battery statistics of the devices are reset before each measurement in order to get accurate results.

The command used to get the battery statistics is:

adb shell dumpsys batterystats --charged <app package name>

Table 5.12: Average battery usage with number of successful runs in parenthesis

Number of test cases	Samsung device (mAh)			Honor device (mAh)		
	Appium	Espresso	Robotium	Appium	Espresso	Robotium
8	8.393(24)	0.634(24)	0.832(23)	16.319(22)	12.279(23)	13.985(21)
12	13.588(43)	1.628(44)	1.931(42)	26.411(42)	16.639(43)	19.377(39)

Initially, eight test cases were run and then finally twelve test cases were run and hence the results of power consumption obtained for both the number are mentioned separately in Table 5.12. Since it would be logical to take only executions where all the test cases were successful, the runs that resulted in some failures of test cases were excluded from the measurement in each framework. The 'Check Element for flakiness' test (T13) was not considered in this measurement due to the unpredictable nature of the test.

It can be observed that Appium has the maximum power consumption in both the devices and Espresso has the least.

5.2 Evaluating Capabilities

After the implementation of all the test cases, there were certain capabilities that all could perform and there are capabilities that some frameworks could perform while others could not. The capabilities found are summarized in the Table below and detailed discussion is given in Chapter 6. If the framework can perform the capability but has certain problems in some cases, it is mentioned as partial and the explanation is mentioned in the comments column.

Table 5.13: Summary of Capabilities of different frameworks

Capability	Appium	Robotium	Espresso	Comments
Selection of child element from parent view	Yes	Yes	Yes	

Selection of particular index of element	Yes	Partial	Yes	Can be done only for certain elements such as text or elements in ListView
Pressing the back button	Yes	Yes	Partial	Does not work on dialogs, popups and root activity in Espresso
Taking Screenshot	Yes	Yes	No	
Waiting for elements to appear	Yes	Yes	Yes	
Entering text in element	Yes	Yes	Yes	
Opening and Interacting with elements in Navigation Drawer	Partial	Partial	Partial	None of the frameworks had the capability of directly interacting with items inside navigation drawer and they had to be identified separately
Scrolling behaviour	Yes	Yes	Partial	The element has to be inside a ScrollView in Espresso
Clicking on particular item in RecyclerView	Yes	Yes	Yes	
Identifying items in Dialogs	Yes	Yes	Partial	Sometimes, the items could not be identified in Espresso
Opening and Interacting with Menu Items	Partial	Yes	Partial	Only Robotium has an inbuilt method to directly identify items inside the menu
Interacting with Soft Keyboard	Yes	Yes	Partial	Espresso can only close the soft keyboard but it cannot perform any other interactions
Performing long click on element	Yes	Yes	Yes	
Interacting with Floating Action Button	Yes	Partial	Yes	Robotium sometimes cannot detect the floating action button
Interacting with items inside the spinner directly	No	Yes	No	
Swiping the page in any direction	Yes	Yes	Yes	
Toggling Wi-Fi connection	Yes	Yes	No	
Identifying Toast Messages	No	Yes	Yes	

Clearing the app preferences and the Database	Yes	Partial	Partial	Knowledge of the source code and data is required in Espresso and Robotium
Interacting with elements outside the AUT	Yes	No	No	
End-to-End Testing of multiple devices	Yes	No	No	

From Table 5.13, it can be observed that there is no single framework that can perform all the capabilities completely. Appium can perform 17 (80.95%) capabilities completely, Robotium can perform 15 (71.4%) capabilities completely and Espresso can perform only 9 (42.86%) capabilities completely. This shows that Appium is the best framework in terms of capabilities while Espresso is the worst.

6. Discussion

In this chapter, the results presented in the previous chapter are discussed in more detail.

6.1 Measurement of KPIs (RQ 2.1)

6.1.1 Execution Time

Execution Time is one of the most important performance indicators as it gives a measurement about how fast the framework can finish the execution of all the test cases. Through the use of the Welch Two Sample t-test, it was found that the differences between mean execution times of each framework are significant. This shows that execution time can be used as a KPI for comparison of test automation frameworks. In both the devices, Espresso was the fastest in all the test cases.

In the Samsung device, there were two test cases where Robotium was slower than Appium. In T7, it was found that Robotium framework took some time in detecting the Floating Action Button and the text in the popups causing Appium to be faster. Floating Action Button is a circular button responsible for triggering a primary action in the mobile app's UI. In T11, Robotium again took time detecting the Floating Action Buttons making it to be slower than Appium. Appium was slower in all the other test cases. From the Boxplots in Figure 5.1, Appium has the largest variation of execution time in all the test cases except T11 where Robotium had the largest variation. This was mainly due to the unpredictability in detection of the floating action button in Robotium causing the framework to detect the element quickly in some cases and detect the element after sometime in other cases. There were also outliers observed in all the frameworks and Appium had the maximum of them. The outliers were present due to the complex flows present in the AUT and the framework detecting the elements at different speeds. The presence of maximum outliers in Appium could be the due to the large variation in execution time which depends on different locator strategies used by Appium to find the elements and due to time constraints, all the possible locator strategies were not studied.

Unlike the outcome for the Samsung device where Appium was faster than Robotium in two test cases, for the Honor device the framework Appium was found to be the slowest in all the test cases as it takes more time to detect the elements as well as execute actions. It can also be observed that the execution time for the Honor device is longer than for the Samsung device for all the test cases and on all the frameworks and this was expected due to the slowness of the Honor device. It can be noticed that there are more outliers in different frameworks for the Honor device compared to the Samsung device. This is because of the speed difference in movement across different screens of the mobile app in the slower device resulting in elements sometimes appearing quicker or sometimes appearing slower. The usage of a complex mobile application with multiple flows plays a significant role in the presence of outliers. Appium again has the maximum number of outliers and the largest variation of the execution time showing the unpredictability of the framework with respect to this KPI.

There are multiple deciding factors for the execution time such as the way the framework handles the UI of the AUT and also the framework's architecture. The instrumentation frameworks – Espresso and Robotium are faster than the non-instrumentation framework - Appium. It can be concluded that in terms of execution time, Espresso is the best framework having the fastest execution time in both the devices and Appium is the worst framework having the slowest execution time in both the devices. Appium is slow at typing text making the time longer when text has to be entered while Espresso and Robotium directly add the text to the UI widget much faster. The overhead that occurs due to the client-server communication in Appium where the Appium servers are responsible for performing actions on the device could also result in increase in execution time. Espresso seems to be much faster in scanning the different elements on the screen, searching for element in list and scrolling across the screen making it the best framework based on this KPI. It also automatically runs test commands at suitable time when the main thread is idle making it more faster in execution [43].

6.1.2 Maintainability

The maintainability was measured in this study based on the change in LOC for different releases and the total LOC at the end for all the frameworks.

In an Agile/DevOps environment, the code of the application is bound to change frequently and some functionality may be affected in some way or other, and thus it is necessary to make modifications in the tests frequently to make them work in the latest version of mobile application. The LOC added, modified or deleted during different releases can be a good indication of how much maintainability is required for the particular framework compared to other frameworks. The best framework should require less changes in LOC per release compared to other frameworks.

Table 5.3 shows the changes in LOC per test cases. From the Table, it can be estimated that the total change in LOC for Appium is 78, for Espresso is 57 and for Robotium is 64. In T1, during different releases, the option of optionally entering the number for verification was implemented as some devices can detect the number automatically and to implement this change, Appium required more changes in LOC due to handling of exceptions when element is not detected. In T2, the entire flow was changed and Appium required more changes in LOC to handle new flow with different elements to be detected. In T3, one of the toggles was removed and Appium required more changes in LOC to avoid detection of that element. For T4, one of the buttons was modified with respect to the id of the element and all the frameworks just required one small change. In T7, one of the floating Action Buttons was modified and it required more changes in LOC in Robotium due to the framework's problem of detecting the floating Action Button in some devices. Overall, Appium requires more changes in LOC compared to the other frameworks. Thus, it can be concluded that based on the change in LOC, Appium is the worst for Maintainability and Espresso is the best. However, it is also to be noted that this depends on the type of change being implemented and the results may vary depending on the change.

Another measurement for maintainability could be the total LOC for setting up the framework and execution of all the test cases. From the Table 5.4, it can be estimated that the total LOC for Appium is 409, for Espresso is 374 and for Robotium is 379. On further research, it was found that the difference in number of wait statements as well as framework's capability to directly perform some actions using inbuilt functions played a major role in the final outcome. Espresso required less wait statements due to the framework's capability of handling synchronization automatically. Also, Espresso could directly perform some actions using inbuilt functions within a single line such as scrolling and performing some action on an element while other frameworks required separate line for scrolling and separate line for performing some action. Appium needed more LOC for setting up the framework due to the usage of drivers and running of the application outside the mobile application's source code while Espresso and Robotium required very less setup.

Since Appium has the largest change in LOC and total LOC and Espresso has the smallest change in LOC and total LOC, it can be concluded that Appium is the worst framework in terms of maintainability and Espresso is the best framework.

6.1.3 Flakiness

Some frameworks have the capability to detect an element quickly whereas some frameworks take time to detect elements especially when the mobile application is complex and involves many flows. This could cause the tests to pass in some cases but fail in others. Therefore, to make the tests pass, it is imperative to add wait statements to the test cases to avoid the test failing during all the cases. This is mainly necessary in slow devices such as the Honor device used in this study. The number of wait statements can be a good indication of the degree of the framework to handle flaky tests as the more wait statements are required, the lesser is the framework's capability to handle flaky tests. The frameworks have different ways of handling the wait statements.

The command used by Robotium is:

***solo**.waitForView(id);*

The id corresponds to the id of the element the framework has to wait for. Another action such as clicking of the element cannot be done in the same command and has to be done separately. No arguments such as the duration of the wait needs to be passed and is handled implicitly by the framework.

The command used by Appium is:

```
new WebDriverWait(driver,10).until(ExpectedConditions.visibilityOfElementLocated(
    By.id("com.example.debug:id/button_reject_call"))).click();
```

The id of the element is passed along with the duration for waiting in seconds (10 in the above example). Different actions such as clicking of the element can be combined in the same command. The above example shows the explicit wait capability of Appium where the duration of wait for each element is specified separately. Appium also supports implicit wait capability where all the elements in a page are made to wait for the same duration until that element is found. This is done by using the command:

```
driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
```

In the case of Espresso, for this study, a custom matcher was used which checks whether the element is present or not and if not present, repeat checking after an interval for few iterations. The commands used are:

```
try {
  check(matches(matcher))
  true
} catch (e: Exception) {
  false
}
```

The matcher in the above code sample is the matcher of the element to be found. The above code sample is repeated for certain iterations until element is found.

From Table 5.5, it can be observed that Appium requires the most number of wait statements while Espresso requires the least. It was found that framework Espresso has the capability to detect elements very quickly even across screens due to the automatic handling of synchronization. For Robotium, though the framework has the capability to detect some elements, it cannot perform any action on it and therefore the wait statements were required in some elements. Appium on the other hand required large number of wait statements as it cannot detect elements quickly. Based on the results, it can be said that Espresso is the best framework in terms of flakiness and Appium is the worst.

One interesting thing to note is that the number of wait statements in Espresso can be reduced to zero through the use of the idling resource capability of Espresso. However, this requires a change in the source code of the application and hence, it was avoided in this study. A custom matcher was used for waiting for a particular element in Espresso. Also, if the AUT was a simpler app without complex hierarchies and flows, then the number of wait statements would have reduced considerably in all the frameworks.

Table 5.6 shows the results of execution of the flaky test (T13) in different frameworks. A total of 112 tests were run, 56 in each device. It can be observed that in the case of Espresso, no tests failed indicating that the element can be detected as soon as it appears and this is due to the automatic handling of synchronization. In Robotium, two tests failed and it was found that this occurred in the slower Honor device and the framework failed to detect the element on time although it was present. In Appium, four tests failed, one of them in the faster Samsung device and three of them in the slower Honor device indicating clearly that the framework takes a long time to detect the element once it appears. It can be concluded that in terms of flakiness, Espresso is the best framework and Appium is the worst. In order to execute flaky tests in Appium framework, it would be necessary to increase the waiting time to make the test pass or implement explicit waits with maximum waiting time which could be measured on multiple runs and this could affect the performance.

6.1.4 Reliability

The main goal of the mobile application testing automation frameworks is to identify any bugs in the required functionality in the mobile application. Therefore, if the tests fail due to some other reasons which are framework specific due to the framework's inability to execute certain functionalities, this could lead to wastage of resources in identifying the reason the test cases failed when there is no defect

in the mobile application. Thus, reliability plays a high role in the performance of the framework. For this study, since the test cases that are implemented do not fail on purpose, then theoretically, all the test cases should pass every time they are run. But issues in the framework could affect the results of the test cases.

From the Table 5.7, it can be observed that all the frameworks have some number of failed tests in the Samsung device. It was found that Appium sometimes failed to scroll the screen resulting in T9 failing the most number of times. The other two tests failed because Appium failed to detect an element resulting in timeout and the reason for that behaviour is unclear as on verifying the screenshot of the failed tests, the element seems to be present on the screen.
In Espresso, two tests failed. The reason T2 failed was due to the usage of animator in one of the button resulting in the test getting stuck for some period of time and then failing. This can be prevented by avoiding usage of animators in elements though other frameworks do not suffer from this problem. The reason T7 failed was due to the framework failing to interact with the elements in the dialog some times. On further research, it was found that Espresso generally has problems interacting with elements on the dialog.
Robotium seems to have the most number of failed test cases. There was one instance where the navigation drawer failed to open resulting in failure of T2. The reason T7 failed was due to the framework not being able to detect the menu item sometimes. The reason T10 failed was due to the framework not being able to scroll down the screen sometimes. T11 failed due to the framework having problems to detect the Floating Action Button Menu items.

All the frameworks have some number of failed tests in the Honor device as well. In Appium, the main test that failed was again T9 due to the failure to scroll the screen as in the Samsung device. The reason T7 failed was failure to detect the menu item. Finally, all the other tests failed due to the issue of Appium sometimes not detecting the element even though it was present which also occurred in the Samsung device.
In Espresso, the same two tests failed as those in Samsung device with the same reasons as described above but this time with more frequency.
Robotium seems to have the most number of failed test cases in the Honor device as well. The reason T2 failed was similar to the issue in the Samsung device regarding opening the navigation drawer. T5 failed sometimes due to the VoIP icon not being able to get detected though the element was present on screen. The reason T7 and T8 failed was similar to the issue in Samsung device with respect to detecting menu items. T9 and T11 failed due to the failure to swipe to bottom of the navigation drawer. Finally, T12 failed due to the element not getting detected in some cases and the reason is unclear.

It can be observed that the frameworks have more failures in the Honor device than in the Samsung device and this can be due to the slowness of the device and failure to execute certain functionalities. Robotium has more failures due to issues with some functionalities such as detecting the Floating Action Button, opening the navigation drawer or detecting the menu items. Appium similarly has some failures mainly due to failure to detect some elements and scrolling of screen. Espresso has some failures due to the issue of usage of animators in elements and it was found that since the framework waits until the main thread is idle, usage of animators will result in the main thread never being idle leading to failure of the test. Robotium has the most number of failures making it less reliable compared to the other frameworks and Espresso seems to be the most reliable framework with the least number of failures.

An issue that was specific to Appium that sometimes affected the reliability of the test cases was the failure of the Appium server. Since Appium has a client-server architecture, it requires the client to send commands to the server which is started on a specific port and listens for requests. Sometimes, this results in a “socket hang up” error on the server side leading to the failure of all the test cases from the time the error occurs and requires restarting of the server and running all the test cases again. On further research, it was found that this could be happen due to multiple reasons, one of them being bugs in the server itself. There are no bullet-proof solutions known yet for this issue and it needs to be kept in mind when using Appium although it happens only a few times. A workaround could be to batch the commands which is a feature supported by Appium where multiple Appium commands can be packed inside a single Appium command to improve reliability of the tests.

6.1.5 Fragmentation

Fragmentation is one of the major problems in executing automated tests in Android applications due to the plethora of different Android devices with varying speed and capabilities that might result in the same test which is passing in one device to fail in the other. The reduction in the code changes made so that the same test can work on different platforms will play a major role in the performance of the framework. The number of sleep statements and alternate executions can give an indication about how much changes need to be made for different devices.

Espresso required the least number of sleep statements (2) and alternate executions (1). In T2, two sleep statements were needed for the Samsung device as it could not interact with element in dialog in two different cases where as Honor device was able to do so. It was found that this happened due to the element being detected too quickly before the full dialog was opened and hence the interaction could not happen. An alternate execution was needed for closing the soft keyboard in the Honor device as the original inbuilt function was not able to close the soft keyboard in certain cases. On a deeper analysis, it was found that this happened due to the usage of animators in the element clicked before keyboard close action is initiated. Soft keyboard is the on-screen keyboard in the mobile device for typing text.

Appium was the second best with four usages of sleep statements and a single alternate execution. In T1, two sleep statements were needed in the Honor device as the two elements were either detected wrongly or not detected at all. On further research, it was found that this happened due to the screen taking a very long time to appear and therefore sleep statement was required even though the framework's wait statement was used. In T13, a sleep statement was required for the Honor device as the framework could not long press an icon without first waiting for it in some cases. On deeper analysis, it was found that the element detection was slow resulting in the need for a sleep statement. A similar scenario happened in the failure of clicking of menu icon on T6 in the Honor device resulting in usage of sleep statement. An alternate execution was required in scrolling down the navigation view as it was found that the original method of using scroll method in UiAutomator inbuilt in Appium did not work in the slower Honor device.

Robotium required the maximum number of sleep statements (5) and alternate executions (3). The usage of sleep statements in T1 is similar to the usage mentioned in the framework Appium for the same test and same Honor device. In T13, one sleep statement was required in the Honor device as a long press to open a dialog resulted in the next element not getting detected in some cases. It was found that this happened due to the unusual behaviour of the framework scrolling down in dialog before detecting the next element which was at the top resulting in failure. A similar situation resulted in the use of another sleep statement in the same test but this time the screen was scrolled down instead of the dialog before detection resulting in failure. In T11, the spinner text was not detected in the Samsung device resulting in usage of a sleep statement. Two alternate executions were needed as the framework could not detect Floating Action Button in the Samsung device resulting in the usage of detection by image instead of a button for those cases. Another alternate execution was needed for entering text in an AutoCompleteTextView element which could not take place in the Honor device.

Based on the above discussion, it can be concluded that the framework Robotium is the worst in terms of fragmentation due to multiple changes required between the fast and slow device while the framework Espresso is the best with least number of changes.

6.1.6 Identification of defects

One of the main goals to execute automation tests is to check the functionality of the mobile application and to identify any defects which were not identified during the process of manual testing. This can be highly useful during the process of regression testing to check whether the old functionality is still working after addition of new functionalities. The ability of the frameworks to identify defects plays a major role in the performance of the framework.

There were two defects which was found by all the frameworks during the execution of all the test cases. One defect was on searching for a particular contact, it was found that the search result did not display the contact in spite of the contact being saved in the particular device. This was found to happen only in the Honor device. On executing this test in all the frameworks, the test failed due to lack of desired element not being present resulting in identification of the defect.

Another defect found was that on searching for a contact and opening the detail view of the contact, the VoIP icon was not visible but if a conversation with the contact is first opened and then the detail view of the contact was clicked, the VoIP icon would be visible. This was found through executing T5 first resulting in the test failing in all the frameworks indicating that VoIP icon was not present and then executing the T4 which resulted in conversation being opened and then finally executing the T5 again which now passes due to the detection of the icon.

Since all the frameworks were able to identify both the defects, all the frameworks were equal in this category. No other defects were identified although it could be possible to identify more provided more test cases were implemented.

6.1.7 Battery Usage

Since real mobile devices were used for testing instead of the emulators, the battery usage that occurs in the mobile device due to execution of the test cases by the frameworks plays an important role in the performance of the framework. Some frameworks may result in larger usage in battery and it was interesting to see which framework has the least battery drain.

The mobile devices have limited battery capacity. In the context of Agile/DevOps environment, since it is required that the devices have to be connected to the system at all times for the pipeline to run, it might lead to issues such as the device explosion due to overheating of the battery especially when there is a high battery usage. One way of handling this issue is through the usage of device farms which are mobile devices hosted on the cloud. They are virtual devices where the test cases could be executed. However, the drawback is that there might be some bugs which may not be identified unless the test is run on a real device.

From Table 5.12, it can be observed that Appium requires the maximum battery consumption in both the devices and Espresso takes the least. It can be said that the battery consumption is proportional to the time of execution of all the test cases in each framework as Appium has a much longer execution time and the battery consumption seems to be much larger as well while Espresso and Robotium have minute differences in execution times and the battery consumption among them seem to have minute differences as well.

The battery usage obtained depends on various factors such as Power usage, Wi-Fi usage and CPU usage as the adb command takes all factors which could account for drain in battery and sometimes the framework could also result in extra usage of resources. For example, it is necessary to initialize drivers separately in Appium which are responsible for sending commands to the mobile device which might result in additional CPU usage and thus extra battery drainage while Espresso and Robotium executions are handled within the framework itself. It is to be noted that there might be different results of the battery usage for Appium if command batching is used although it might still not be as less as the battery usage in Robotium and Appium. The overhead that occurs due to the client-server communication in Appium could also play a factor in higher power consumption. The type of user interactions such as Tap or Back button performed in the test cases executions could also play a factor in the energy consumption obtained as mentioned in the study [44]. The study also found Espresso to be the most energy efficient.

Thus, it can be concluded that in terms of battery consumption, Espresso is the best framework and Appium is the worst.

6.1.8 Ranking of frameworks

Based on the analysis done on all the KPIs, the frameworks can be ranked to indicate the best and the worst frameworks with respect to the KPIs.

Table 6.1: Framework score based on KPI

	Execution Time	**Maintainability**	**Flakiness**	**Reliability**	**Fragmentation**	**Battery Usage**	**Total**
Appium	3	3	3	2	2	3	**16**

Espresso	1	1	1	1	1	1	**6**
Robotium	2	2	2	3	3	2	**14**

Table 6.1 indicates the scoring given to all the frameworks. If the framework is best in the particular KPI, it is given a score of one and if it is the worst, then a score of three is given. The best framework is the one having the lowest score. The KPI 'Identification of Defects' was not considered since all the frameworks have the same rank. As observed, Espresso being the best in all the categories has the least score of six while Robotium is the next best with a score of fourteen followed by Appium with a score of sixteen.

6.2 Evaluating Capabilities (RQ 2.2)

The capabilities found during the execution of the test cases on all the frameworks are discussed below and code snippets on how they can be executed on different frameworks are displayed in Appendix B.

1. **Selection of child element from parent view**
   It was found that in Espresso, it was possible to select elements under particular parent views using 'childAtPosition' method. In Robotium, there was also a specific method to specify child view of an element directly by using the ViewGroup class. In Appium, it was possible by using 'findElement' method on the parent element. The child element can also be found in Appium by using XPATH which is a locator strategy for identifying element based on the location of the element on a view hierarchy. However it might not be completely reliable as the view hierarchy changes between different mobile devices and using XPATH locator will result in the same test failing in a different device.

2. **Selection of particular index of element**
   It was found that it was possible to specify the index of a particular element being searched using id or text in the frameworks - Espresso and Appium. This has several advantages such as selecting elements which all have same id but at different positions in the view. However in Robotium, it was possible to select an index only of specific type of element such as Text or elements in a list.

3. **Pressing the back button**
   It is possible to navigate back in all the three frameworks but the function does not work on any dialogs, popups and on the root activity in the case of Espresso. This is a major limiting factor in Espresso as most of the mobile apps require back button on dialogs or popups and the Espresso back function just freezes the app indefinitely during such a case causing tests to fail.

4. **Taking Screenshot**
   Robotium and Appium provide inbuilt functions that are capable taking screenshots and storing at specific locations but Espresso does not have any inbuilt functions and relies on some third party library to implement this functionality. However, it is to be noted that the user has to give permission for the application to write into storage space of the mobile device in the case of Robotium [45].

5. **Waiting for elements to appear**
   All the three frameworks provide functions to wait for an element until it is displayed and this is useful in the case of flaky tests. The methods of implementation is similar to the code snippets described in the flakiness section - 6.1.3.

6. **Entering text on element**
   All the three frameworks had the capability of entering text however in Robotium, the inbuilt function had the capability to enter text only on EditText type element not even on alternate implementation of EditText. Therefore, an alternative method had to be used on elements with other types.

It was found that Espresso and Robotium had the capability of the text being replaced with the new inputted text while in Appium, a separate function had to be written to clear the text before entering a new text else the new text gets appended to the already present text.

7. **Opening and Interacting with elements in Navigation Drawer**
   Navigation Drawer is a UI panel that shows the mobile app's main navigation menu. All the three frameworks had the capability of opening the Navigation Drawer. While Espresso and Robotium had inbuilt functions to perform the action, Appium had to get the element using accessibility id and then click on it to open the Navigation Drawer. However, none of the frameworks had the capability of directly interacting with elements inside the Navigation Drawer and the elements had to be identified separately by text.

8. **Scrolling behaviour**
   All the three frameworks had the capability of scrolling up or down, but in Espresso, this was possible only if the view was inside a ScrollView. This was a major bottleneck as in smaller devices, it was not possible to scroll down directly.

9. **Clicking on particular item in RecyclerView**
   RecyclerView is a common list view using in Android. While Espresso and Robotium have inbuilt functions for performing this capability, in Appium, XPATH locator had to be used to get the item in RecyclerView.

10. **Identifying items in Dialogs**
    While Robotium and Appium can always identify the elements in a dialog, Espresso had some difficulties in identifying the elements in certain cases causing the tests to fail. Robotium and Appium could identify the items in the same way as the other items while Espresso had to use a different implementation for identifying these items.

11. **Opening and Interacting with Menu Items**
    Opening the menu tab was possible in all the three frameworks but the option of directly interacting with an item was available only in Robotium while in Appium and Espresso, the element had to be identified separately by text.

12. **Interacting with a Soft Keyboard**
    While all the frameworks support the option of closing the soft keyboard, only Robotium and Appium can interact with it directly by typing some text with it. Espresso does not have the capability of typing the text directly using the soft keyboard.

13. **Performing a long click on an element**
    All the frameworks were capable of performing a long click. While Espresso and Robotium had an inbuilt method which does not require the user to specify the duration of the click, Appium required the user to manually specify the duration of the click in terms of milliseconds.

14. **Interacting with a Floating Action Button**
    Espresso and Appium were able to identify the floating action button always but in the case of Robotium, the framework was not able to identify the button in certain cases leading to usage of alternative implementations such as clicking on a particular index of an image. The implementation is similar to clicking of an element and only the alternate implementation in Robotium is shown below.

15. **Interacting with items inside the spinner directly**
    Spinner is a drop down like menu that is used to display multiple values from which the user can select only one value. Only Robotium has the capability of directly interacting with the item inside the spinner through inbuilt functions while no such capability was found for Appium and Espresso and the interaction with the spinner in these two frameworks can be done by manual means by clicking on the spinner and then finding the element by text.

16. **Swiping the page in any direction**
    All three frameworks have the capability to swipe the page in any direction through different methods, though there is an observable difference in the performance between the different frameworks on performing such an action.

17. **Toggling Wi-Fi connection**
    Robotium and Appium have inbuilt methods that can be used to enable or disable the Wi-Fi connection although permissions will be needed to make this work. Espresso does not have any capability to perform this action. It is to be noted that in the latest android version (10), apps are not allowed to toggle the Wi-Fi state directly and the inbuilt methods will not work only in this version.

18. **Identifying Toast Messages**
    A toast message is the popup dialog that appears for a short span of time displaying a message. Espresso and Robotium were able to identify the toast messages but Appium couldn't do so in spite of the default Toast class present in XPATH locator class. The alternate way is to use OCR to check whether the toast message was displayed.

19. **Clearing the App Preferences and Database**
    Sometimes it would be necessary to clear the app data before the execution of the test cases. Appium provides an inbuilt flag to enable clear the app data but in the other two frameworks, it would be necessary to know the source code and data used to implement this capability which might not be possible in all the cases. In Appium, this is done by setting the "noReset" capability of the driver to false.

20. **Interacting with elements outside the AUT**
    In some cases, it might be necessary to navigate outside the AUT to perform some actions. For example, it might be necessary to enable some option in the settings of the mobile device. It is stated in [46] that Espresso cannot interact with UI elements outside the app. Similarly in [47], it is stated that Robotium can only interact with views that belong to your application. Appium was able to interact with elements outside AUT due to the UiAutomator framework integrated to it. Thus, all the system popups and other functions such as opening notification could only be performed in Appium. To give an example, T14 was written only in Appium which involved adding a contact to the mobile device's native address book. The contact was added by navigating to the native contact application of the device and then performing required actions to add and save a contact.
    Appium performs the required actions through the usage of identifying id of the native element and the use of XPATH locator. While Appium has the capability to do this, it has some limitations. For example, the same code cannot be used in all the mobile devices as the native applications are strongly dependent on the device manufacturer. Thus, a test case written for Samsung device has to be modified for the Honor device. Moreover, the test might also not work with different devices from the same manufacturer if XPATH locator is used due to the change in view hierarchy in different devices.

21. **End-to-End Testing of multiple devices**
    Sometimes it might be necessary to execute a certain test simultaneously on two different devices but we would want to execute different portions of it on different devices. For example, suppose it has to be checked whether the call happens from one device to another. Although it can be done by running two different tests in two different devices, it will not simulate a real-life scenario where the waiting should happen only after the call is placed on one device. Appium provides the capability to do this by executing the one test, but different portions of it in two separate devices. To indicate this, T15 was done in Appium. This is through the creation of two separate Appium drivers for the two mobile devices. In the other two frameworks, the same test or different tests can be executed concurrently on different devices but not different portions of the same test.

Appium seems to be capable of performing the maximum percentage of capabilities considered completely while Espresso has the lowest percentage. The collected statistics are presented in Figure 6.1.

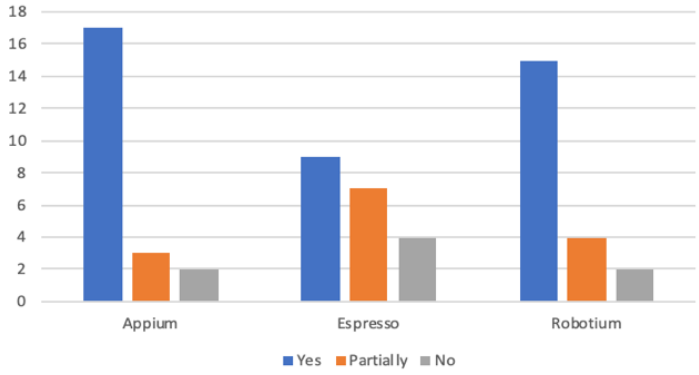


Figure 6.1: Number of capabilities met by the selected frameworks

7. Conclusions and Future work

This section describes the conclusions of our study conducted along with their limitations and possible future work.

7.1 Conclusions

In this study, the comparison of the mobile application testing automation frameworks was conducted in an Agile/DevOps environment. This is one of the first attempts to compare different testing frameworks based on the KPIs identified by literature research, and implementing the study on a complex mobile application and in an Agile/DevOps environment to measure the performance of the frameworks in a real-world setting. Several Android UI testing frameworks were identified that are available in the market today by looking through research papers and blogs. Then the most popular frameworks were identified to perform the study, and since there were no past papers or blogs that particularly mention the popularity of frameworks, they were identified through Google Trends and Stack overflow data. The frameworks which are most popular currently are – Appium, Espresso and Robotium. The structure of the most popular frameworks have been discussed with Espresso and Robotium having similar architecture being part of the Instrumentation framework and Appium having a client-server architecture.

In order to have metrics for comparison of these frameworks, the KPIs for the mobile application testing automation frameworks were identified. The identification was done based on the general performance metrics used in software development along with the specific metrics that are used for testing. The KPIs identified were – execution time, maintainability, flakiness, reliability, fragmentation, identification of defects and battery usage. Then, the challenges of integrating the test automation frameworks into the Agile/DevOps environment were identified to give the reader an idea of what challenges to expect and how to possibly overcome them. The challenges identified were - Unstable test cases, lack of proper infrastructure, distributed organization, build time and finally, awareness and visibility of build and test results.

To collect the data for performing empirical analysis, a few test cases were selected from the complex mobile application used based on the AQuA guidelines and which can give a good measurements of the performance of the frameworks. Then a CI/CD pipeline was built after addressing the challenges of integrating the test automation frameworks into an Agile/DevOps environment. Data was then collected by running different tests using the pipeline at different intervals on both the connected mobile devices.
Based on the analysis, several conclusions could be obtained as discussed in Chapter 6. Espresso is the best framework with respect to all the KPIs analysed. This was mainly because of the automatic synchronization and ability to execute actions quickly and more reliably. Robotium was worse than Appium in the KPIs – Reliability and Fragmentation. This was because of the unreliability in executing certain actions in Robotium and because of large number of code changes required for different devices. Appium was the worst framework in all the other KPIs and thus overall, it was the worst framework. Many factors could be identified, but the main reason was the client-server architecture resulting in slow execution of commands. It could also be due to the reason that the main purpose of Appium is to be used for functional and end-to-end testing of multiple devices for mimicking the real life end user experience.

However, on evaluating the capabilities of the frameworks based on the test cases implemented, Appium was the best framework. This is because of the ability to execute certain capabilities such as end-to-end testing of multiple devices and testing outside the AUT, which other frameworks were not able to do. Moreover, Appium also could perform capabilities such as clearing the app preferences and the database which other frameworks could only perform partially. Espresso on the other hand could perform very less capabilities entirely, thus making it the worst framework in terms of capabilities.

The study we have conducted will be useful for industry to decide the best type of framework for automation of testing, based on their specific needs and ideas of how to possibly minimize the challenges of integration of the frameworks in an Agile/DevOps environment. The different KPIs and capabilities identified will give the users an idea of the type of categories to consider before selecting a particular framework. It is not possible to identify one framework as the best framework overall as each framework has its own strong and weak points. The test automation framework selection needs to be

based on the nature and requirements of the Android application used for testing. If the AUT is simple, then Espresso could be used for test automation as it is the best framework in terms of performance. On the other hand, if the application is very complex and requires capabilities such as end-to-end testing of multiple devices, Appium is the best option due to its large capabilities. Robotium could be used if the application is slightly complex and performance is an important requirement.

7.2 Limitations of this study

There were several difficulties faced in this study. Firstly, implementing the test cases in a complex mobile application was not easy due to the nature of the mobile app involving different types of elements and complex flows. Therefore, a deeper research had to be conducted into each framework to explore all the possible ways in which some action could be performed. Secondly, it was difficult to come across information about the test automation frameworks and their capabilities in recognised scientific literature and we had to go through numerous blogs and discussions to gather this information.

There are several limitations to this study. Firstly, only the most popular open-source frameworks were considered in this study and we did not consider any paid or subscription based frameworks. Another limitation is that only the Android test automation frameworks were considered, without other platforms such as iOS. Thirdly, there might be a limitation of not considering all the capabilities of different frameworks due to time constraints such as the cross-platform capability of Appium. Next, the setup of the environment could also play a factor in the data collected from the KPIs. For example, the execution time might change for each framework in each test case if the setup environment changes either through the use of different CI/CD tools for the pipeline or through complete change in environment. Also, KPIs such as Fragmentation could have more accurate results provided more than two mobile devices were used. Next, the number of challenges associated with integration of the test frameworks into an Agile/DevOps environment might be minimal and more challenges could be identified provided more test cases were implemented and the continuous deployment principle was included. Finally, the study was conducted only on one complex mobile application due to the time constraints of executing all the test cases in different frameworks.

7.3 Future work

This study could be extended in the future to more complex applications with more test cases to further evaluate the different KPIs and capabilities. More mobile application testing automation frameworks could be used for comparison including paid frameworks as well. The list of KPIs and capabilities along with the challenges in Agile/DevOps environment could be extended to include new ones or to add more details to the existing ones. Finally, testing in iOS and other platforms could be conducted using more cross-platform test automation frameworks.

7.4 Reflections

This thesis project gives guidance to the mobile application developers and testers when planning to execute automated tests on mobile devices. The project performs a comparison of the three most popular mobile application testing automation frameworks in an Agile/DevOps environment along with the challenges due to the integration of the frameworks in such an environment. The comparison performed on different test automation frameworks should give a clear idea of which framework to select based on the requirement and functionality of the mobile application being tested along with ways to address the challenges in an Agile/DevOps environment.

As the complexity of mobile applications increases, different corporations will benefit from the automation of testing to deliver the best product to the end users. When working on this project, only internal phone numbers were used and no user data was collected and the app was only run on two specific devices used for this study.

It is our belief that this study will benefit many organizations in the problem of choosing which mobile application testing automation framework to use available in the market today, as they can have a better idea about the performance and capabilities of each framework since this study was conducted on a complex mobile application that reflects a real-world industrial setting.

Bibliography

[1] J. Clement, "Google Play Store Numbers," 2020. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ . [Accessed 27 February 2020].

[2] J. Gao, X. Bai, W.-T. Tsai and T. Tsai, Mobile Application Testing :A Tutorial, IEEE Computer Society, 2014.

[3] S. Zein, N. Salleh and J. Grundy, "A systematic mapping study of mobile application testing techniques," *The Journal of Systems and Software,* 2016.

[4] S. Nidhra and J. Dondeti, "BLACK BOX AND WHITE BOX TESTING TECHNIQUES –A LITERATURE REVIEW," *International Journal of Embedded Systems and Applications (IJESA),* vol. 2, no. 2, 2012.

[5] R. Meier, Professional Android 4 Application Development, John Wiley & Sons, 2012.

[6] H. Muccini, A. D. Francesco and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in *7th International Workshop on Automation of Software Test (AST), IEEE*, Zurich, 2012.

[7] M. Kropp and P. Morales, "Automated GUI Testing on the Android Platform," *On Testing Software and Systems: Short Papers,* p. 67, 2010.

[8] A. Debbiche, M. Dien´er and R. B. Svensson, "Challenges When Adopting Continuous Integration: A Case Study," in *International Conference on Product-Focused Software Process Improvement, Springer*, Cham, 2014.

[9] M. Shahin, M. A. Babar and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," in *IEEE Access 5*, 2017.

[10] C. Merina, . N. Anggraini and N. Hakiem, "A Comparative Analysis of Test Automation Frameworks Performance for Functional Testing in Android-Based Applications using the Distance to the Ideal Alternative Method," in *Third International Conference on Informatics and Computing (ICIC), IEEE*, Palembang, 2018.

[11] A. M. Sinaga, P. A. Wibowo, A. Silalahi and N. Yolanda, "Performance of Automation Testing Tools for Android Applications," in *10th International Conference on Information Technology and Electrical Engineering, IEEE*, Kuta, 2018.

[12] K. S. Arif and U. Ali, "Mobile Application testing tools and their challenges: A comparative study," in *2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET), IEEE*, Sukkur, 2019.

[13] L. Ardito, R. Coppola, M. Morisio and M. Torchian, "Espresso vs. EyeAutomate: An Experiment for the Comparison of Two Generations of Android GUI Testing," *Proceedings of the Evaluation and Assessment on Software Engineering,* pp. 13-22, 2019.

[14] P. Tramontana, D. Amalfitano, N. Amatucci and A. R. Fasolino, "Automated functional testing of mobile applications: a systematic mapping study," *Software Quality Journal 27.1,* pp. 149-201, 2019.

[15] A. R. Chowdhury and P. Krishnakumar, "Mobile Application Testing using Automation frameworks," July 2019. [Online]. Available: https://www.browserstack.com/guide/mobile-application-testing-frameworks. [Accessed 28 February 2020].

[16] A. Zhukovich, "Android UI Testing Frameworks," 10 September 2019. [Online]. Available: https://proandroiddev.com/android-ui-testing-frameworks-b0b52187ceb. [Accessed 28 February 2020].

[17] T. Lämsä, "Comparison of GUI testing tools for Android applications," University of OULU, 2017.

[18] "Google Trends," [Online]. Available: https://trends.google.com/trends/. [Accessed 1 March 2020].

[19] "Stack Overflow," [Online]. Available: https://stackoverflow.com/. [Accessed 1 March 2020].

[20] "StackExchange," [Online]. Available: https://data.stackexchange.com/ . [Accessed 1 March 2020].

[21] Paul, "A Deconstruction of the Appium Architecture," Edureka, 22 May 2019. [Online]. Available: https://www.edureka.co/blog/appium-architecture/ . [Accessed 14 April 2020].

[22] H. Benake, "HB Blog 136: Android Automation Testing Using Espresso Framework," 19 May 2017. [Online]. Available: http://harshalbenake.blogspot.com/2017/05/hb-blog-136-android-automation-testing.html. [Accessed 14 April 2020].

[23] "Test UI for a single app," [Online]. Available: https://developer.android.com/training/testing/ui-testing/espresso-testing.html . [Accessed 14 April 2020].

[24] R. Coppola, L. Ardito and . M. Torchian, "Fragility of layout-based and visual GUI test scripts: an assessment study on a hybrid mobile application," in *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2019.

[25] R. Coppola, M. Morisio and M. Torchiano, "Mobile GUI Testing Fragility: A Study on Open-Source Android Applications," *IEEE TRANSACTIONS ON RELIABILITY ,* vol. 68, 2019.

[26] Q. Luo, F. Hariri, L. Eloussi and D. Marinov, "An empirical analysis of flaky tests," *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering,* pp. 643-653, 2014.

[27] M. Linares-Vásquez, K. Moran and D. Poshyvanyk, "Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing," *International Conference on Software Maintenance and Evolution (ICSME), IEEE,* 2017.

[28] K. Herzig, M. Greiler, J. Czerwonka and B. Murphy, "The Art of Testing Less without Sacrificing Quality," *IEEE International Conference on Software Engineering,* 2015.

[29] P. Hegde, "5 Incredibly Useful KPIs for Test Automation," 9 September 2019. [Online]. Available: https://www.logigear.com/blog/test-automation/5-incredibly-useful-kips-for-test-automation/. [Accessed 2 March 2020].

[30] . H. Kim, B. Choi and S. Yoon, "Performance testing based on test-driven development for mobile applications," in *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*, 2009.

[31] A. Debbiche, M. Dienér and R. B. Svensson, "Challenges when adopting continuous integration: A case study," in *International Conference on Product-Focused Software Process Improvement, Springer*, Cham, 2014.

[32] A. Nilsson, J. Bosch and . C. Berger, "Visualizing Testing Activities to Support Continuous Integration: A Multiple Case Study," *International Conference on Agile Software Development (XP),* 2014.

[33] J. D. Blischak, E. R. Davenport and G. Wilson, "A Quick Introduction to Version Control with Git and GitHub," *PLoS Comput Biol.,* 2016.

[34] P. Rai, M. S. Dhir, M. and A. Garg, "A prologue of JENKINS with comparative scrutiny of various software integration tools," *2nd International Conference on Computing for Sustainable Global Development (INDIACom), IEEE,* 2015.

[35] "Baseline Testing Criteria for Android Applications," 30 November 2014. [Online]. Available: https://www.appqualityalliance.org/AQuA-test-criteria-for-android-apps. [Accessed 7 April 2020].

[36] Rajkumar, "How to generate Extent Reports in Selenium Webdriver," 2019 23 October. [Online]. Available: https://www.softwaretestingmaterial.com/generate-extent-reports/. [Accessed 30 March 2020].

[37] "Viewing Battery Use Data," 20 04 2016. [Online]. Available: https://www.cnblogs.com/pengdonglin137/articles/5411802.html. [Accessed 04 05 2020].

[38] A. Ghahrai, "Test Automation Tips and Best Practices," 16 April 2020. [Online]. Available: https://devqa.io/automation/test-automation-tips-best-practices/ . [Accessed 20 April 2020].

[39] T. Ylonen, "The Secure Shell (SSH) Authentication Protocol," Cisco, January 2006. [Online]. Available: https://www.hjp.at/doc/rfc/rfc4252.html. [Accessed 20 April 2020].

[40] S. Elbaum, G. Rothermel and J. Penix, "Techniques for improving regression testing in continuous integration development environments," *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering,* 2014.

[41] "Blue Ocean for Jenkins Pipelines," [Online]. Available: https://jenkins.io/projects/blueocean/. [Accessed 20 April 2020].

[42] B. BEERS, "P-Value Definition," 19 February 2020. [Online]. Available: https://www.investopedia.com/terms/p/p-value.asp. [Accessed 6 May 2020].

[43] J. Unadkat, "Appium vs Espresso: Key Differences," 13 March 2020. [Online]. Available: https://www.browserstack.com/guide/appium-vs-espresso. [Accessed 10 May 2020].

[44] L. Cruz and R. Abreu, "On the Energy Footprint of Mobile Testing Frameworks," *IEEE Transactions on Software Engineering,* 2019.

[45] S. Joshi, "15+ Useful Robotium Code Snippets for Android Test Automation," 11 June 2014. [Online]. Available: https://www.javacodegeeks.com/2014/06/15-useful-robotium-code-snippets-for-android-test-automation.html. [Accessed 4 May 2020].

[46] "Stackoverflow:Questions- press overview button in Android Espresso," 15 March 2017. [Online]. Available: https://stackoverflow.com/questions/42819656/press-overview-button-in-android-espresso/42822723. [Accessed 4 May 2020].

[47] "Stackoverflow Questions: Robotium Test Case For Android (System Dialog can not find OK and Cancel Buttons)," 13 December 2013. [Online]. Available: https://stackoverflow.com/questions/20613740/robotium-test-case-for-android-system-dialog-can-not-find-ok-and-cancel-buttons. [Accessed 6 May 2020].

Appendix A: Detailed Steps of Test Cases

T1

The steps executed in this test case are -

- Step 1: Click “Get Started” Button on Home Page
- Step 2: Click Accept Permission Description Dialog
- Step 3: Enter mobile number
- Step 4: Click “Okay” Button
- Step 5: Click “Agree” Button on Privacy Page
- Step 5: Click “Type Name Manually” Button
- Step 6: Enter First Name
- Step 7: Enter Last Name
- Step 8: Click “Continue” Button
- Step 9: Click “Allow” Button for Draw Over Other Apps Permission
- Step 10: Click “Later” Button on Skip Backup Page
- Step 11: Click “Confirm” Button
- Step 12: Click Back to Dismiss Popup

T2

The steps executed in this test are -

- Step 1: Click on the navigation icon to open navigation drawer
- Step 2: Click edit profile icon on navigation drawer page
- Step 3: Click on “Create Business Profile” button
- Step 4: Click “Continue” Button
- Step 5: Enter business name
- Step 6: Choose “Education” option
- Step 7: Choose “College” option
- Step 8: Click “Next” Button
- Step 9: Enter the pin code
- Step 10: Click “Next” Button
- Step 11: Click “Enter Manually” Button
- Step 12: Enter street address
- Step 13: Click “Finish” Button
- Step 14: Click “I’m Done” Button
- Step 15: Click “Save”
- Step 16: Dismiss the Navigation Drawer

T3

The steps executed in this test are –

- Step 1: Click on the navigation icon to open navigation drawer
- Step 2: Click on “settings” option in navigation drawer page
- Step 3: Click on “general” option in settings page
- Step 4: Scroll to last element while reading the state of all toggles
- Step 5: Activate the last toggle
- Step 6: Scroll to the top
- Step 7: Scroll to the bottom again reading all the state of toggles

- Step 8: Check whether the count of all checked toggles increased by one

T4

The steps executed in this test are -

- Step 1: Click on the floating action button on homepage
- Step 2: Enter contact to send message in search field
- Step 3: Select the contact from the list
- Step 4: Enter the message
- Step 5: Click on the send button
- Step 6: Click on the navigation button to navigate back to home page

T5

The steps executed in this test are-

- Step 1: Click on the home icon in bottom bar
- Step 2: Click on the search field in home page
- Step 3: Click back to dismiss the popup dialog that appears
- Step 4: Enter contact name in search field
- Step 5: Select first contact from list
- Step 6: Click back to dismiss the popup dialog that appears
- Step 7: Click on the VoIP icon
- Step 8: Wait for five seconds and disconnect call

T6

The steps executed in this test are -

- Step 1: Click on the floating action button on homepage
- Step 2: Enter contact to send message in search field
- Step 3: Select the contact from the list
- Step 4: Click on the menu button at right hand side corner
- Step 5: Click “block Contact” option
- Step 6: Click “block” button
- Step 7: Click “close” button on popup dialog
- Step 8: Click on the spam tab
- Step 9: Search for the contact and click on it
- Step 10: Click on the menu button at right hand side corner
- Step 11: Click “unblock Contact” from the displayed list
- Step 12: Click “yes” button on popup dialog
- Step 13: Click on personal tab and check whether contact is displayed

T7

The steps executed in this test are –

- Step 1: Click on the floating action button on homepage
- Step 2: Click on “Create group chat” button
- Step 3: Select one contact from list
- Step 4: Select another contact from list
- Step 5: Click on the next button

- Step 6: Enter group name
- Step 7: Click on tick mark button
- Step 8: Click on the top toolbar in conversation
- Step 9: Click on edit icon
- Step 10: Enter a new group name
- Step 11: Click on okay button
- Step 12: Click on “Leave group” option
- Step 13: Click on “leave group” in confirmation dialog

T8

The steps executed in this test are –

- Steps 1-8 are the same as the steps 1-8 in T6 (for blocking contact one)
- Steps 9-16 are the same as the steps 1-8 in T6 (for blocking contact two)
- Step 17: Long click on contact one in spam tab
- Step 18: Click on contact two
- Step 19 : Click on the menu button at right hand side corner
- Step 20: Click “mark as unread” option
- Step 21: Check whether the number of unread conversations is increased by two
- Steps 22-29 are the same as the steps 9-12 in T6 (for unblocking contact one)
- Steps 30-37 are the same as the steps 9-12 in T6 (for unblocking contact two)

T9

The steps executed in this test are –

- Steps 1-8 are the same as the steps 1-8 in T6 (for blocking contact)
- Step 9: Click on the navigation icon to open navigation drawer
- Step 10: Click settings on navigation drawer page
- Step 11: Click on “block” option in settings page
- Step 12: Scroll to “my block list” option and click it
- Step 13: Check whether blocked contact is present
- Step 14: Click on the subtraction(-) icon next to the contact
- Step 15: Click “YES” on the popup dialog

T10

The steps executed in this test are –

- Step 1: Click on the navigation icon to open navigation drawer
- Step 2: Click settings on navigation drawer page
- Step 3: Click on “block” option in settings page
- Step 4: Click on “update” button in the block settings page
- Step 5: Wait until the required element appears

T11

The steps executed in this test are –

- Step 1: Click on the navigation icon to open navigation drawer
- Step 2: Click settings on navigation drawer page
- Step 3: Click on “block” option in settings page
- Step 4: Scroll to “unlock premium” button and click on it

- Steps 5-10: Wait until required element is displayed and swipe right , repeat for all the 6 elements

T12

The steps executed in this test are –

- Step 1: Click on the navigation icon to open navigation drawer
- Step 2: Click on “settings” option in navigation drawer page
- Step 3: Click on “block” option in settings page
- Step 4: Click on the floating action button
- Step 5: Click on “Block a number” floating action button menu item
- Step 6: Click on spinner and select country code
- Step 7: Enter the number to block
- Step 8: Click “Block Button”
- Step 9: Click on Spam Tab
- Step 10: Check whether number is present and click on it
- Step 11: Click on the menu button at right hand side corner
- Step 12: Click “unblock Contact” from the displayed list
- Step 13: Click “yes” button on popup dialog

T13

The steps executed in this test are –

- Step 1: Click on the navigation icon to open navigation drawer
- Step 2: Click on “settings” option in navigation drawer page
- Step 3: Click on “general” option in settings page
- Step 4: Scroll to last element and activate the toggle
- Step 5: Click back to go to previous page
- Step 6: Click “Privacy Center” option
- Step 7: Click Deactivate option
- Step 8: Click Yes on the popup dialog
- Steps 9 – 20: Follow the steps in T1 for activation
- Step 21: Click on the navigation icon to open navigation drawer
- Step 22: Click on “settings” option in navigation drawer page
- Step 23: Click on “general” option in settings page
- Step 24: Scroll to last element and check whether toggle is in deactivated state

T14

The steps executed in this test are -

- Step 1: Click on the contacts icon in bottom bar
- Step 2: Click on the add icon to add new contact
- Step 3: Enter name of contact in native contact application
- Step 4: Enter mobile number of the contact
- Step 5: Click on the save button
- Steps 6-10: Same as steps 1-5 in T5
- Step 11: Click on the menu button at right hand side corner
- Step 12: Click on remove contact option from displayed list
- Step 13: Click “yes” button on popup dialog

T15

The steps executed in this test are -

Device 1 -

- Steps 1-8 are the same as the steps 1-8 in T6 (for blocking contact)

Device 2 -

- Steps 1-5 are the same as steps 1-5 in T5
- Step 6: Click on the Call icon
- Step 7: Click the end call button

Device 1 -

- Step 9: Click on the spam tab
- Step 10: Verify that the blocked call message from contact is present

Appendix B: Code Snippets of execution of capabilities in different frameworks

1. **Selection of child element from parent view**

   Espresso:
   *onView(allOf(withId(*R.id.***pageNextBtn***), *childAtPosition(withId(*R.id.***pageNextBtn***),*
   0), isDisplayed()));

   Robotium:
   ViewGroup viewGroup = (ViewGroup) ***solo****.getView(R.id.****pageNextBtn****,0);*
   viewGroup.getChildAt(0);

   Appium:
   MobileElement parent = ***driver****.findElement("****com.example.debug:id****" +*
   *"****/pageNextBtn****"));*
   *parent.findElement(MobileBy.id(****"com.example.debug:id/pageNextBtn"***
   ));

2. **Selection of particular index of element**

   Espresso:
   *onView(withIndex(withId(R.id.****pageNextBtn****), 0));*

   Robotium:
   solo*.clickOnText(****"text"****,0);*

   Appium:
   driver*.findElements(By.id(****"com.example.debug:id/pageNextBtn"****)).get(0).click();*

3. **Pressing the back button**

   Espresso:
   Espresso.pressBack();

   Robotium:
   solo*.goBack();*

   Appium:
   driver*.pressKey(****new*** *KeyEvent().withKey(AndroidKey.****BACK****));*

4. **Taking Screenshot**

   Robotium:
   solo*.takeScreenshot();*

   Appium:
   driver*.getScreenshotAs(OutputType.****FILE****);*

5. **Entering text on element**

   Espresso:
   *onView(withId(R.id.****edit****)).perform(replaceText(****"Some Text"****));*

   Robotium:
   solo*.enterText((EditText)****solo****.getView(R.id.****edit****),****"Some text"****);*

Appium:
driver.findElement(By.id("com.example.debug:id/edit")).sendKeys("Some Text");

6. Opening and Interacting with elements in Navigation Drawer

Espresso:
onView(withId(R.id.navigation_view)).perform(DrawerActions.open());
onView(withId(R.id.block_settings)).perform(click());

Robotium:
solo.clickOnActionBarHomeButton();
solo.clickOnView(solo.getView(R.id. block_settings));

Appium:
driver.findElementByAccessibilityId("Navigate up").click();
driver.findElement(By.id("com.example.debug:id/block_settings")).click());

7. Scrolling behaviour

Espresso:
onView(withId(R.id.scroll_view)).perform(scrollTo());

Robotium:
solo.scrollToBottom();

Appium:
driver.findElementByAndroidUIAutomator("new UiScrollable(new UiSelector()).scrollIntoView" + "(new UiSelector().resourceId" + (\"com.example.debug:id/scroll_view\""");

8. Clicking on particular item in RecyclerView

Espresso:
onView(withId(R.id.recyclerView)).perform(RecyclerViewActions.actionOnItemAtPosition(0, click()));

Robotium:
solo.clickInRecyclerView(0, 0); //Clicking on first item in first RecyclerView

Appium:
driver.findElementsByXPath("//android.support.v7.widget.RecyclerView[0]").get(0).click();

9. Identifying items in Dialogs

Espresso:
onView(withText("YES")).inRoot(isDialog()).check(matches(isDisplayed())).perform(click());

10. Opening and Interacting with Menu Items

Espresso:
openActionBarOverflowOrOptionsMenu(InstrumentationRegistry.getTargetContext());
onView(withText(containsString("Block"))).perform(click());

Robotium:
solo.clickOnMenuItem("Block");

Appium:
```
driver.findElementByAccessibilityId("More options").click();
driver.findElementByAndroidUIAutomator("new
UiSelector().textContains(\"Block\")").click();
```

11. Interacting with a Soft Keyboard

Espresso:
```
Espresso.closeSoftKeyboard();
```

Robotium:
```
solo.hideSoftKeyboard();
solo.sendKey((int)'A' - 36); //For typing A using soft keyboard
```

Appium:
```
driver.hideKeyboard();
driver.pressKey(new KeyEvent(AndroidKey.BUTTON_A)); //For typing A using  soft
keyboard
```

12. Performing a long click on an element

Espresso:
```
onView(withId(R.id.messagesTabIcon)).perform(longClick());
```

Robotium:
```
solo.clickLongOnView(solo.getView(R.id.messagesTabIcon));
```

Appium:
```
new TouchAction(driver).TouchAction(driver).longPress(
LongPressOptions.longPressOptions().withElement(ElementOption.element(driver.findE
lement(By.id("com.example.debug:id/messagesTabIcon")))).waitAction(WaitOpti
ons.waitOptions(Duration.ofMillis(500))).perform().release();
```

13. Interacting with a Floating Action Button

Robotium:
```
solo.clickOnImage(12); //12 is the index of image representing floating action button
```

14. Interacting with items inside the spinner directly

Robotium:
```
solo.pressSpinnerItem(0, 0); //Pressing first item in the first spinner
```

15. Swiping the page in any direction

Espresso:
```
onView(withText("Some Text")).perform(swipeRight());
```

Robotium:
```
solo.drag(0,10,0,10,2); //Swipe from (0,0) to (10,10)
```

Appium:
```
new TouchAction(driver).press(PointOption.point(0,0)).moveTo
((PointOption.point(10, 10)).release().perform(); ); //Swipe from (0,0) to (10,10)
```

16. Toggling Wi-Fi connection

Robotium:
```
solo.setWiFiData(true);
```

Appium:

```
driver.setConnection(new ConnectionStateBuilder().withWiFiEnabled()
.build());
```

17. Identifying Toast Messages

Espresso:

```
onView(withText("Toast text")).inRoot(withDecorView(not(new
ActivityTestRule<>(Activity.class).getActivity().getWindow()
.getDecorView()))).check(matches(isDisplayed()));
```

Robotium:

```
solo.searchText("Toast Text");
```

18. Clearing the App Preferences and Database

Appium:

```
new DesiredCapabilities().setCapability("noReset", "false");
```

19. End-to-End Testing of multiple devices

Appium:

Calling from the first device:

```
driver.findElement(By.id("com.example.debug:id/call_icon"))
.click();
```

Ending call from second device:

```
driver2.findElement(By.id("com.example.debug:id/end_call"))
.click();
```

TRITA-EECS-EX-2020:690