Federal University of São Carlos– UFSCar
Center for Exact Sciences and Technology– CCET
Department of Computing– DC
Graduate Program in Computer Science– PPGCC

Pedro Henrique Kuroishi

Automated Testing of Mobile Applications: Methodological Foundations and a Practical Device Farm Solution

São Carlos
2025

Pedro Henrique Kuroishi

Automated Testing of Mobile Applications: Methodological Foundations and a Practical Device Farm Solution

Ph.D. Thesis presented to the Graduate Program in Computer Science of the Center for Exact Sciences and Technology at the Federal University of São Carlos, as part of the requirements for obtaining the title of Doctor in Computer Science.

Area of concentration: Methodologies and Techniques of Computing

Advisor: Prof. Dr. Auri Marcelo Rizzo Vincenzi
Co-advisor: Prof. Dr. José Carlos Maldonado

São Carlos
2025

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Defesa de Tese de Doutorado do candidato Pedro Henrique Kuroishi, realizada em 26/11/2025.

Comissão Julgadora:

Prof. Dr. Auri Marcelo Rizzo Vincenzi (UFSCar)

Prof. Dr. André Takeshi Endo (UFSCar)

Prof. Dr. Valter Vieira de Camargo (UFSCar)

Prof. Dr. Rui Filipe Lima Maranhão de Abreu (U.Porto)

Prof. Dr. Renato de Freitas Bulcão Neto (UFG)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

Acknowledgements

First, I would like to thank my family for all the support throughout this journey. In all the difficult moments, you encouraged me and helped me keep going.

A special thank you to Professor Dr. Auri Marcelo Rizzo Vincenzi and Professor Dr. José Carlos Maldonado, for their guidance, friendship, and teaching throughout this work.

Finally, I would like to thank CAPES (Grant nº 001 and 88887.888653/2023-00) and CNPq (Grant nº 141137/2021-5 and 140435/2025-5) for their financial support.

Abstract

Context: Software testing activities are essential in the software development life cycle since they help to find possible bugs before releasing the product for end users. Despite its importance, testing is still not widely applied when considering the mobile ecosystem. When testing a mobile application, it is essential to consider how to execute the tests (i.e., techniques, methods, approaches, tools, and frameworks) and where the tests run (i.e., on real devices, emulators, or in the cloud). Over the years, researchers and the industry have proposed various infrastructures to facilitate test execution. However, choosing the most suitable testing infrastructure depends on the application's scope, supported features, and the company's/user's needs. Moreover, there is still a need to explore ways to optimize testing for mobile applications.

Objective: This thesis aims to provide an implementation of a solution capable of running tests on multiple devices, and these tests should validate the communication between the application and an external IoT device. Moreover, the second goal is to subsidize the testing process by providing an approach for test case generation and quality assessment. In both cases, we considered two hardware components of an Android device: Bluetooth and Location.

Method: This thesis presents a different methodology comprising a collection of published and submitted work from the PhD, providing distinct contributions aligned with the defined research goals.

Conclusion: During this PhD, we made solid contributions by providing means to enhance mobile application testing, focusing on the Android ecosystem. In this thesis, we presented four papers published in renowned Software Engineering journals and conferences, and one paper in the submission process. We expect that the results of this work will benefit not only the academic community but also practitioners.

Keywords: mobile testing;mobile application testing; Android; infrastructure; testing

infrastructure; device farm; local device farm; Bluetooth; Location; test generation; test case generation; automated test generation; LLM; large language models; mutation testing; systematic studies; tertiary study; mapping study; academia-industry.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Context, Motivation, and Conceptualization

Software products are everywhere. Over the years, software has increasingly been integrated into society to facilitate the resolution of various problems quickly and efficiently, creating an interdependence between software and society. Another factor that helps this dependence is the constant technological advances that facilitate this human-machine interaction. The evolution from desktop to mobile devices allowed users to access software from anywhere. In this sense, it is increasingly mandatory for developers to deliver reliable, high-quality software products.

It is common knowledge that testing is fundamental to ensuring certain levels of quality in the software product being developed, as it can reveal defects that arise during development (MYERS; SANDLER; BADGETT, 2011). In summary, testing consists of selecting inputs from the software input domain, executing them in the program under test, and analyzing the results by verifying whether the output matches the expected output. When exploring the presented concept, one may find that testing is straightforward. However, several factors can hinder not only the execution of the test activity but also its significant adoption.

In general, testing activities are expensive and time-consuming. Several studies in the academic literature estimate that testing activities consume, on average, 50% of the total cost of software development (HARROLD, 2000; KARHU et al., 2009). In this sense, automating the software testing process can bring many benefits. For instance, automated tests may reduce test execution time, reduce the human effort to perform

repeated tasks, and minimize possible errors when conducting testing activities (KARHU et al., 2009; CERVANTES, 2009). Moreover, several studies found that test automation could improve software product quality (KARHU et al., 2009; WANG et al., 2022).

Regardless of how the tests will be performed (manually or automatically), applying the activities from the beginning of the software development cycle is always essential and cheaper. The study of Boehm e Basili (2001) shows that the cost of fixing a bug in software in production can be up to 100 times higher if the bug were found in the early development stages. Therefore, the sooner a defect is found, the lower the cost of fixing it (BOEHM; BASILI, 2001).

The last few years have been marked by massive technological advancements, particularly a significant increase in the number of mobile device users. By 2025, the number of mobile devices operating worldwide will be estimated to surpass 18.22 billion (STATISTA, 2022). Considering that the world population is around 8 billion, we can calculate a ratio of 2 mobile devices per inhabitant[1].

This popularity may be explained by the ubiquity of mobile devices, which allow people to perform multiple daily activities on a single smartphone. It can be used for communication, entertainment, shopping, banking, gaming, and tracking health status. Some applications leverage mobile device sensors to provide services and information to users (ALMEIDA; MACHADO; ANDRADE, 2020). For instance, Uber[2] provides location-based transportation services. Strava[3] uses GPS to track workouts such as walking, running, or cycling, and it also connects with different devices (e.g., smartwatches) to collect helpful information. Moreover, the concepts of *Industry 4.0* (GHOBAKHLOO, 2020) e *Smart Cities* (YIN et al., 2015) also transformed the way people interact with the environment, making mobile devices crucial in this new paradigm (MATYI et al., 2020).

A mobile application (or simply app) is designed to run on mobile devices (e.g., smartphones, tablets, smartwatches, and so on). It takes in input the user's interactions or contextual information (MUCCINI; FRANCESCO; ESPOSITO, 2012). In general, there are three types of mobile applications (GAO et al., 2014; SANTOS; FILHO; SOUZA, 2020; HALLER, 2013; FARIA, 2019; MENEGASSI; ENDO, 2016): native, web-based, and hybrid.

According to Gao et al. (2014), mobile app testing refers to *"testing activities on mobile devices using well-defined software test methods and tools to ensure quality in functions, behaviors, performance, and quality of service, as well as features, such as mobility, usability, interoperability, connectivity, security, and privacy"*.

Although testing is a fundamental activity in the software development life cycle, mobile application testing has several aspects that differ from other traditional applications,

[1] It is essential to emphasize that this is a rough comparison intended to illustrate the massive adoption of mobile devices, without accounting for socioeconomic factors.

[2] https://www.uber.com/br/pt-br/

[3] https://www.strava.com/

such as desktop and client-server applications (MUCCINI; FRANCESCO; ESPOSITO, 2012). Mobile applications are designed to run on mobile devices, thus, one have to consider all the diverse characteristics and limitations of the devices, such as different types of connectivity (e.g., Wi-Fi, Bluetooth, NFC, 4G, 5G), screen size and density, multiple sensors (e.g., GPS, accelerometer, pedometer, gyroscope), limited battery and hardware resources. Moreover, some applications use contextual information to provide services to users, e.g., mobile context-aware applications (ALMEIDA; MACHADO; ANDRADE, 2020).

When testing a mobile application, the developer/tester should also consider where the tests will run, i.e., which devices/emulators will be used to execute them. One may think testing on all available devices is the most obvious option. However, the market offers many devices of different models and brands. Moreover, to increase the complexity of this equation, one must also consider the device's operating system. This phenomenon, known as *fragmentation*, makes testing even more difficult, as testing in all possible configurations of mobile devices is generally impossible (WEI; LIU; CHEUNG, 2016).

In 2022, Android dominated the mobile operating systems with over 70% of the market share (STATCOUNTER, 2022). According to Figure 1, more than 10 versions of Android are available, and each can be adapted and customized by different vendors to meet their needs. Figure 2 also illustrates the impacts of fragmentation by showing the number of existing Android devices in 2015. In this case, more than 24,000 devices of different types and models were available. Although this data is outdated, it is clear that it is infeasible given the financial cost of acquiring all the devices and the time spent testing an application in all possible configurations; i.e., testing in all possible combinations is generally impossible (WEI; LIU; CHEUNG, 2016). Hence, researchers must focus on exploring various techniques and approaches to ensure the quality of applications and provide a better experience for end users.

When testing a mobile application, the environment (or infrastructure) where the tests are executed must also be considered. In this case, Gao et al. (2014) defines four mobile test infrastructures: **emulation-based**, **device-based**, **cloud-based**, and **crowd-based**.

The emulation-based approach involves creating a device simulator, i.e., a virtual machine that simulates the characteristics of a mobile device, such as screen size, OS version, and device model. This approach is considered cheap because the simulators can be created on a personal computer. However, these simulators are limited in features, and depending on application requirements, they may not be the best option. For instance, an Android emulator does not support Bluetooth connections.

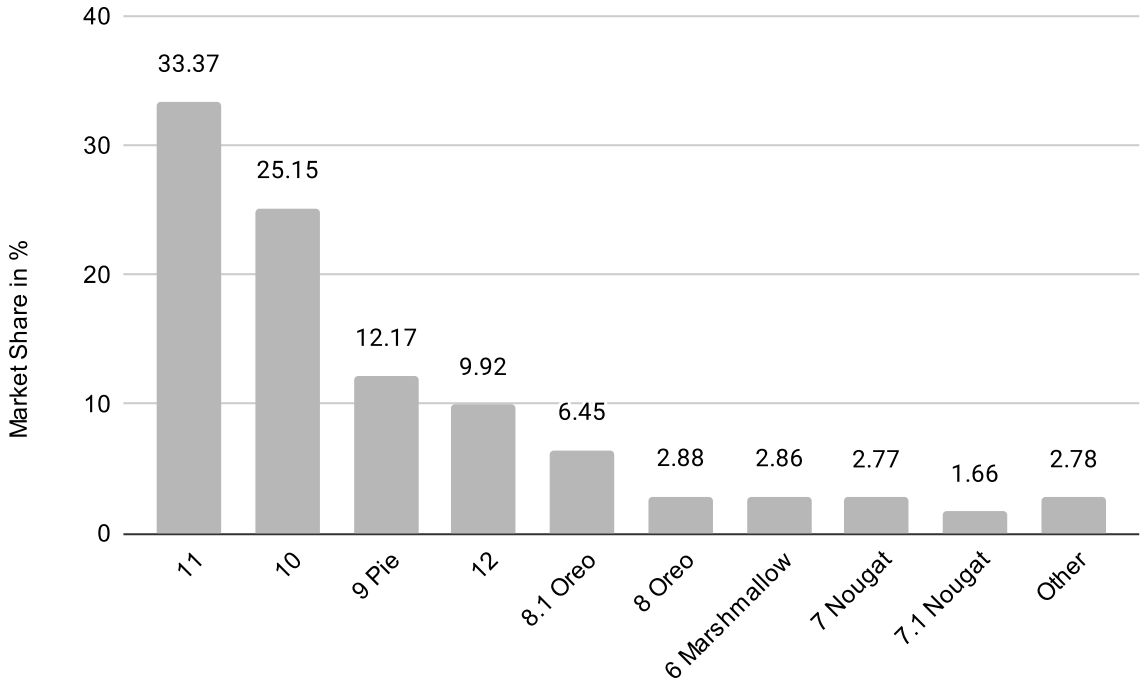


Figure 1 – Global mobile Android version market share in 2022 (extracted from StatCounter (2022)).

The second approach is device-based. It involves performing tests on real mobile devices. This provides a more realistic testing scenario than emulators because the tests run on a real device and have access to all its functions. However, it may be a costly alternative due to the need to acquire real devices.

The cloud-based approach involves building a mobile device cloud to support large-scale testing. In general, this approach uses a pay-as-you-go business model in which the user has access to all devices (emulators or real devices) and, in exchange, pays for the minutes used/executed tests. AWS Device Farm (AMAZON, 2024), Google Firebase Test Lab (LLC, 2024), and Visual Studio App Center (CORPORATION, 2024) are some examples of mobile device cloud services. Despite the advantages related to its cost-effectiveness for testing large-scale applications (GAO et al., 2014; ROJAS; MEIRELES; DIAS-NETO, 2016), these services still suffer from security issues related to vulnerabilities and the fact that the cloud infrastructure can interfere with the test runs' results (FAZZINI; ORSO, 2021).

The crowd-based approach uses freelancers/contracted testers/users or the community to perform the tests under real-world conditions (ZHANG; GAO; CHENG, 2017). In general, the advantages are scalability, low cost, and the possibility of testing in a real scenario (LIU; ZHANG; CHENG, 2019). On the other hand, the approach may yield low-quality tests, as they are generally executed *ad hoc*.

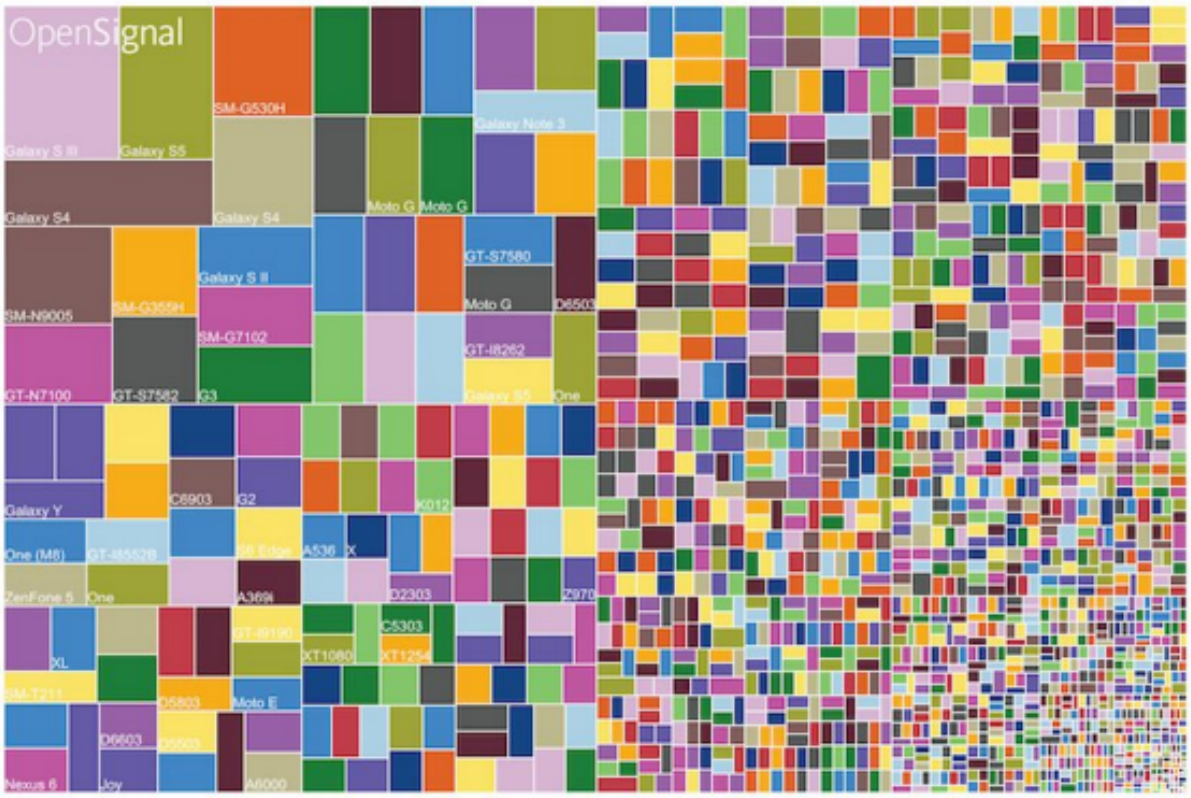


Figure 2 – Illustration of mobile device fragmentation (Android OS) in 2015.

The use of an infrastructure facilitates the execution of tests on multiple devices (LANUI; CHIEW, 2019), and tends to reduce fragmentation impacts. However, the choice of infrastructure depends on several factors, such as application requirements and the features it offers.

Another common activity during the testing process consists of devising the input used to validate an application under test. In general, tests can be performed manually or through automation (RAMLER; WOLFMAIER, 2006; HAAS et al., 2021; TAIPALE et al., 2011). Despite their advantages and disadvantages, both practices can be applied to enhance product quality, and choosing one over the other is not straightforward and depends on various aspects of a project under development (TAIPALE et al., 2011). This study focused on the automation aspects of software testing.

Automated test case generation has been widely explored in academia (POTUZAK; LIPKA, 2023; WANG et al., 2021; ZENG et al., 2016; RAMLER; KLAMMER; BUCHGEHER, 2018; CHOUDHARY; GORLA; ORSO, 2015). For Java unit test case generation, EvoSuite (FRASER; ARCURI, 2011) has been broadly recognized as the state-of-the-art tool for many years (HERLIM et al., 2021; VOGL et al., 2021; SCHWEIKL; FRASER; ARCURI, 2023). In the context of mobile, the majority of studies aim to generate auto-

mated test cases/scripts to validate the GUI aspects of an application under test (KONG et al., 2019; AMALFITANO et al., 2015; NIE et al., 2023; CHOUDHARY; GORLA; ORSO, 2015).

Recently, advances in large language models (or simply LLMs) have revolutionized the scientific field, providing a new paradigm for future research. LLMs are language models based on the Transformer architecture (VASWANI et al., 2017) capable of producing human-like text due to their efficiency in natural language processing tasks (KASNECI et al., 2023; RAIAAN et al., 2024), pervading various areas of knowledge.

LLMs are also adopted for Software Engineering tasks such as code generation, code summarization, code translation, code evaluation, code management, vulnerability detection, and question-and-answer interactions (ZHENG et al., 2024). Specifically in Software Testing, the academic literature has made considerable effort to explore the capabilities of existing LLMs to enhance test case automation (WANG et al., 2025; WANG et al., 2024).

Regardless of the approach adopted for automated test case generation, metrics are still needed to estimate the quality of the generated tests. Code coverage is a commonly adopted metric and provides information regarding statements, branches, and paths exercised during the test execution (ELBAUM; GABLE; ROTHERMEL, 2001). However, code coverage may be misleading since there is still no consensus on the effectiveness of the metric in detecting faults (GOPINATH; JENSEN; GROCE, 2014; INOZEMTSEVA; HOLMES, 2014; HEMMATI, 2015; KOCHHAR; THUNG; LO, 2015).

Mutation testing is a fault-based testing technique used to assess the quality of a given test suite (DEMILLO; LIPTON; SAYWARD, 1978). The mutation testing process can be simplified as follows: (i) generate faulty versions of the original application under test by seeding artificial defects modeled by mutation operators. Next, (ii) the tests are executed against each mutant. Then, (iii) the adequacy is computed by the ratio of killed mutants to non-equivalent ones. Despite being an effective metric for assessing test suite quality, this technique still needs improvement due to its inherent cost, which impedes its widespread practical use (PIZZOLETO et al., 2019).

In the context of mobile applications, there are some initiatives to enable mutation testing (SILVA et al., 2022). However, given the complexity of the mobile ecosystem, it is necessary to explore mutation testing, given the inherent characteristics of this type of application.

The present study focuses on Android application testing due to its open-source nature and the prevalence of studies on this operating system.

The Android tests run locally or on an Android device (i.e., an emulator of a real device) (Android Developers, 2023). Local tests are usually run on a local machine, which is often small and fast. Tests that run on an Android device are called instrumented tests. In this case, the instrumented tests have access to the Android framework API,

enabling testing of specific components, such as UI components (e.g., buttons, text fields). However, these tests tend to be slower than local tests because they require installation on an emulator or a real device. Moreover, there are two types of instrumented tests: instrumented unit tests and instrumented UI tests.

Instrumented unit tests leverage the Android API components but do not interact with the UI. For instance, some mobile device properties can be tested to check whether Bluetooth is on. JUnit[4] and Roboletric[5] are examples of tools to support instrumented unit tests. Instrumented UI tests also leverage Android API components and interact with the UI. For instance, tests to validate the UI state in response to user actions (keyboard input, button clicks, swipes, etc.). Google Espresso[6] and Appium[7] are examples of tools to support instrumented UI tests.

1.2 Hyothesis

This thesis hypothesizes that it is feasible to define a dedicated testing infrastructure that improves the testing process for mobile applications that interact with IoT devices, guided by an automated testing framework and strategies for generating relevant test scenarios.

1.3 Objectives

Although mobile application testing, specifically on Android, has evolved over the years, there are few contributions from the scientific community to investigate and propose mechanisms to validate applications that use Android device hardware components such as connectivity (e.g., Bluetooth, NFC, and Wi-Fi), location, and sensors (e.g., gyroscope, accelerometer, and pedometer). Therefore, introducing new solutions based on these components can benefit academia and industry.

This work is being conducted under the MAI/DAI program (in Portuguese, Programa de Mestrado e Doutorado Acadêmico para Inovação), in partnership with Centro de Pesquisas Avançadas Wernher von Braun (or Von Braun Labs). Currently, the company faces a significant challenge: a lack of testing infrastructure to run mobile application tests across multiple devices. Moreover, the applications developed by Von Braun Labs interact with a specialized IoT device using Bluetooth and Location components, which provide different services to end users.

Therefore, the main goal of this is to provide means for enhancing the testing process of the company in a twofold view:

[4] <https://junit.org/junit5/>
[5] <https://robolectric.org/>
[6] <https://developer.android.com/training/testing/espresso>
[7] <https://appium.io/>

- ❏ **(G1): Propose an investigation and implementation of a solution capable of running tests on multiple devices, and these tests should validate the communication and interaction between the application and the IoT device.**
- ❏ **(G2): Propose means to enhance the testing process by providing Android automated test case generation and quality assessment solutions, considering Bluetooth and Location components.**

The rationale for G1 is to provide the company with a solution to validate its product and improve its quality. The rationale for G2 is to provide a basis for improving the testing process by investigating the use of LLMs to generate test cases for the components mentioned above, and using mutation testing as the baseline to assess the quality of these test cases.

To guide this thesis, the following research questions were defined:

- ❏ **(RQ1): What are the characteristics of a testing infrastructure that validates the communication and interaction of a mobile application and an IoT device?**
- ❏ **(RQ2): How can test case generation be enhanced to target Android Bluetooth and Location components?**

1.4 Methodology and Thesis Structure

1.4.1 Overview

The present thesis is structured as a collection of papers published and submitted during the PhD. Due to the PPGCC-UFSCar program's regulations, the papers are presented in the format required for thesis presentation. That is, it is presented in a different format from the original. However, we emphasize that the content of each paper is preserved in its entirety.

Four papers were published in different venues, and one is currently being submitted to a special issue. Additionally, there is information about two co-authored papers during the PhD. Below, we present information on each paper, sorted in chronological order of publication, providing the title, authors, year of publication, type of venue (Journal or Conference), venue name, status (published or submitted), Qualis/CORE/Quartile[8], and DOI (when available):

- ❏ **First Author:**

[8] This metric was collected from known sources: <https://ppgcc.github.io/discentesPPGCC/pt-BR/qualis/>, <https://www.scimagojr.com/>, and <https://portal.core.edu.au/conf-ranks/>

- **Title:** Towards the definition of a research agenda on mobile application testing based on a tertiary study
  - **Authors:** Pedro Henrique Kuroishi, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi
  - **Year:** 2023
  - **Type of Venue:** Journal
  - **Venue Name:** Information and Software Technology
  - **Qualis/CORE/Quartile:** A1 / A / Q1
  - **Status:** Published
  - **DOI:** <https://doi.org/10.1016/j.infsof.2023.107363>
- **Title:** Towards the Implementation of a Mobile Application Testing Infrastructure at Von Braun Labs
  - **Authors:** Pedro Henrique Kuroishi, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi
  - **Year:** 2023
  - **Type of Venue:** Conference
  - **Venue Name:** IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)
  - **Qualis/CORE/Quartile:** A3 / A / -
  - **Status:** Published
  - **DOI:** <https://doi.org/10.1109/ISSRE59848.2023.00078>
- **Title:** Testing Infrastructure to Support Mobile Application Testing: A Systematic Mapping Study
  - **Authors:** Pedro Henrique Kuroishi, Ana Cristina Ramada Paiva, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi
  - **Year:** 2024
  - **Type of Venue:** Journal
  - **Venue Name:** Information and Software Technology
  - **Qualis/CORE/Quartile:** A1 / A / Q1
  - **Status:** Published
  - **DOI:** <https://doi.org/10.1016/j.infsof.2024.107573>
- **Title:** Designing Mutation Operators for Android Device Components: A View Through Bluetooth and Location API's
  - **Authors:** Pedro Henrique Kuroishi, Ana Cristina Ramada Paiva, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi

  - **Year:** 2025
  - **Type of Venue:** Conference
  - **Venue Name:** XXXIX Simpósio Brasileiro de Engenharia de Software
  - **Qualis/CORE/Quartile:** A3 / - / -
  - **Status:** Published
  - **DOI:** <https://doi.org/10.5753/sbes.2025.9878>

- **Title:** Automated Android Instrumented Unit Test Generation using Large Language Models for Bluetooth and Location Components: An Experimental Study
  - **Authors:** Pedro Henrique Kuroishi, Ana Cristina Ramada Paiva, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi
  - **Year:** 2025
  - **Type of Venue:** Journal
  - **Venue Name:** Empirical Software Engineering
  - **Qualis/CORE/Quartile:** A1 / A / Q1
  - **Status:** Submitted
  - **DOI:** –

❑ **Co-author**:

- **Title:** Static and Dynamic Comparison of Mutation Testing Tools for Python
  - **Authors:** Lucca Renato Guerino, Pedro Henrique Kuroishi, Ana Cristina Ramada Paiva, and Auri Marcelo Rizzo Vincenzi
  - **Year:** 2024
  - **Type of Venue:** Conference
  - **Venue Name:** XXIII Simpósio Brasileiro de Qualidade de Software
  - **Qualis/CORE/Quartile:** B1 / - / -
  - **Status:** Published
  - **DOI:** <https://doi.org/10.1145/3701625.3701659>

- **Title:** METFORD – Mutation tEsTing Framework fOR anDroid
  - **Authors:** Auri MR Vincenzi, Pedro H Kuroishi, João Bispo, Ana RC da Veiga, David RC da Mata, Francisco B Azevedo, Ana CR Paiva
  - **Year:** 2025
  - **Type of Venue:** Journal
  - **Venue Name:** Journal of Systems and Software

* **Qualis/CORE/Quartile:** A2 / A / Q1
* **Status:** Published
* **DOI:** <https://doi.org/10.1016/j.jss.2024.112332>

1.4.2 Thesis Structure

In this thesis, each chapter, except for Chapter 1 (Introduction) and Chapter 7 (Conclusion), presents an article published or submitted during the PhD. Figure 3 depicts the structure of the thesis. Additionally, we assign an ID to each article to make it easier to reference throughout the thesis.

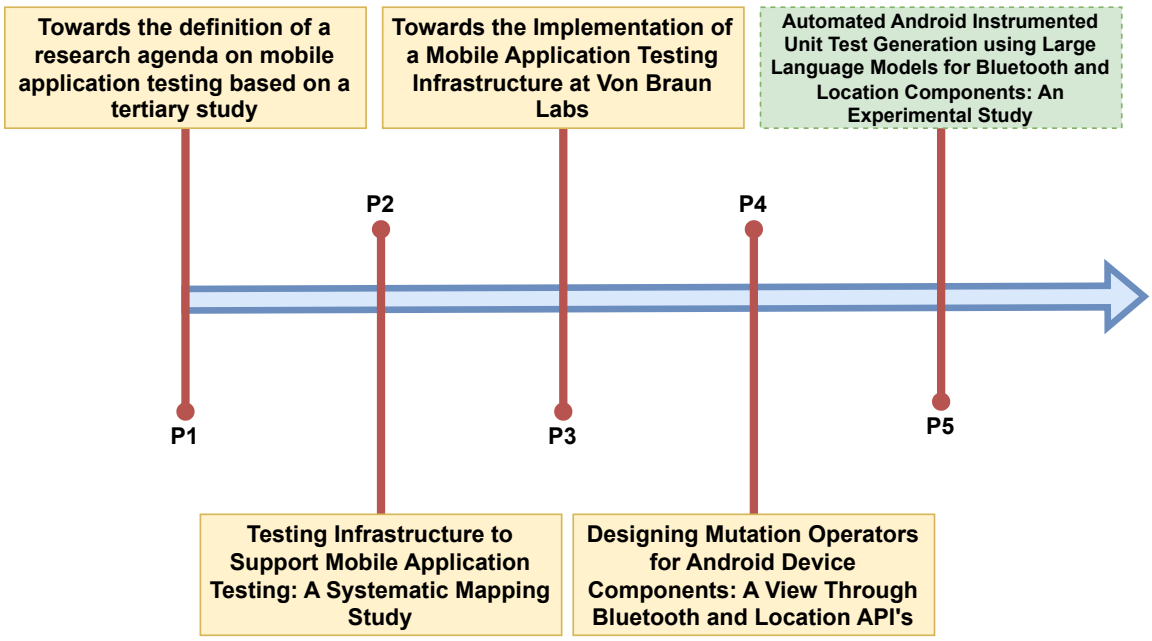


Figure 3 – Paper published/submitted during the PhD.

We followed the same methodology of Kudo et al. (2022). In this paper, the authors reported their experience using systematic studies, such as tertiary reviews, systematic reviews, and mapping studies, to identify current research gaps and guide the research proposal.

It all started with a tertiary study (P1), whose main objective was to collect evidence from existing secondary studies and guide the definition of the PhD research proposal. In summary, a tertiary review consists of a review of secondary studies (KITCHENHAM et al., 2010). We provided details on how the tertiary study was conducted in Chapter 2. After identifying a research topic, we conducted a more fine-grained secondary study (P2) through a mapping study (PETERSEN; VAKKALANKA; KURNIARZ, 2015). This study helped us to have a broad overview of the evidence on the research topic.

The results of the two systematic reviews provided sufficient evidence to conduct the first specific study on the research topic, presented in P3. In this paper, we compare existing testing infrastructure and define a local device farm to run tests across multiple devices within a CI/CD pipeline.

Next, we advanced the research by investigating ways to optimize the testing process. For this matter, we conducted two different studies: P4 and P5. For P4, we proposed a set of mutation operators for Bluetooth and Location components of the Android API. This work started during a PhD "sandwich", which I have the opportunity to attend at the University of Porto under the supervision of Prof. Dr. Ana Cristina Ramada Paiva. Therefore, the mutation operator proposed in P4 extends the operators previously devised in METFORD (VINCENZI et al., 2025). In P5, we investigated devising unit instrumented test cases and used the mutation operators from P4 to validate the results.

In summary, using this methodology (KUDO et al., 2022) facilitated the conduct of this PhD by providing a broad overview of what would be required over the four years. In this case, each paper serves as a checkpoint for the initial plan defined by the candidate and the advisor. We knew we would encounter difficulties, especially passing the reviewers' scrutiny. Furthermore, we always feel that more could have been done to improve the work. However, we are confident that the results of this thesis will offer positive perspectives and contributions to the scientific community.

Chapter 2

Towards the Definition of a Research Agenda on Mobile Application Testing based on a Tertiary Study

2.1 Overview

This chapter presents the first paper published during the PhD. In summary, the study presents a tertiary study that structured the existing secondary studies (i.e., systematic literature reviews, systematic mapping study, and literature surveys) in the field of mobile application testing. The results provided a basis for understanding the research opportunities and defining a research agenda, considering the open challenges. Below, the information regarding the paper is presented (see Figure 4):

- **Title**: Towards the definition of a research agenda on mobile application testing based on a tertiary study.
- **Authors**: Pedro Henrique Kuroishi (UFSCar), José Carlos Maldonado (ICMC-USP) and Auri Marcelo Rizzo Vincenzi (UFSCar).
- **Local**: Information and Software Technology.
- **Year**: 2023.
- **Status**: Published.
- **DOI**: <https://doi.org/10.1016/j.infsof.2023.107363>.

Figure 4 – Information regarding paper P1.

2.2 Abstract

Context: Mobile application testing has gained considerable attention in recent years since mobile devices have become increasingly present in our lives. Unlike traditional software, mobile application testing has to deal with peculiarities, such as screen size and densities, different operating systems, and multiple sensors that increase the complexity of testing.

Objective: This paper summarizes and analyzes the current secondary studies on mobile application testing through a tertiary study.

Method: We selected and analyzed 21 secondary studies related to mobile application testing.

Results: We categorized 21 secondary studies according to their main and specific research topics, test objectives, and testing platforms. Furthermore, we analyze 87 gaps and challenges identified by the secondary studies to understand which gaps have already been addressed and which gaps are still uncovered.

Conclusion: Based on the results, we propose a research agenda with 15 open challenges related to mobile application testing to help future research.

2.3 Introduction

In the past years, the market for mobile devices has rapidly grown. According to STATISTA (2022), the number of mobile devices may surpass 15 billion in 2023. A reasonable explanation for this popularity is the possibility to perform daily tasks (e.g., financial transactions, shopping, and entertainment) from a single smartphone.

In the third quarter of 2022, Google Play[1], the official app store for Android, distributed over 3.4 billion mobile applications (STATISTA, 2022). Similarly, in Apple App Store[2], the official app store for iOS systems, over 2.2 billion applications are available for download (STATISTA, 2022). Given the massive adoption of mobile devices, the quality of an application may impact its success or failure. In this context, testing activities play an essential role in the quality assurance of mobile applications.

In general, mobile application testing differs from testing traditional software (i.e., desktop applications and web applications) because the first has to deal with several mobile features such as connectivity, limitation of resources, multiple sensors, screen size, and density (MUCCINI; FRANCESCO; ESPOSITO, 2012). Furthermore, many applications have to process contextual information to provide their services, increasing the complexity of testing (ALMEIDA; MACHADO; ANDRADE, 2020).

Fragmentation of hardware and software is also a significant challenge related to mobile application testing (JOORABCHI; MESBAH; KRUCHTEN, 2013). Fragmentation occurs due to a heterogeneous ecosystem of mobile devices and operating systems. In 2022, Android and iOS dominate the mobile operating systems with over 99% of the market share (Figure 5), and Android leads the global market with over 70% of users (STATCOUNTER, 2022). According to Figure 6, there are more than ten versions of Android OS available, and each version can be adapted and customized by different vendors according to their needs. Given a heavily fragmented ecosystem, testing an application in all possible mobile devices and OS combinations is generally impossible (WEI; LIU; CHEUNG, 2016). Hence, researchers have to focus on exploring different techniques and approaches to assure the quality of an application and provide a better experience for end-users.

The number of studies tackling different problems related to mobile application testing has increased in the last years (TRAMONTANA et al., 2019; KONG et al., 2019). In this sense, the initial plan was to conduct a secondary study focusing on mobile application testing to guide the definition of the Ph.D. proposal (KUDO et al., 2022). However, after a quick and informal literature search, we found some secondary studies tackling different topics related to mobile application testing.

Therefore, to understand how the area has evolved and the future research directions, we propose a tertiary study aiming to summarize and categorize the secondary studies related to mobile application testing.

A tertiary study is described as a review of secondary research (KITCHENHAM; CHARTERS, 2007). It follows the same procedure as a systematic literature review (KITCHENHAM; CHARTERS, 2007), i.e., planning, conducting the review, and reporting the results. Tertiary studies are not very common in computer science litera-

[1] https://play.google.com/store
[2] https://www.apple.com/app-store/

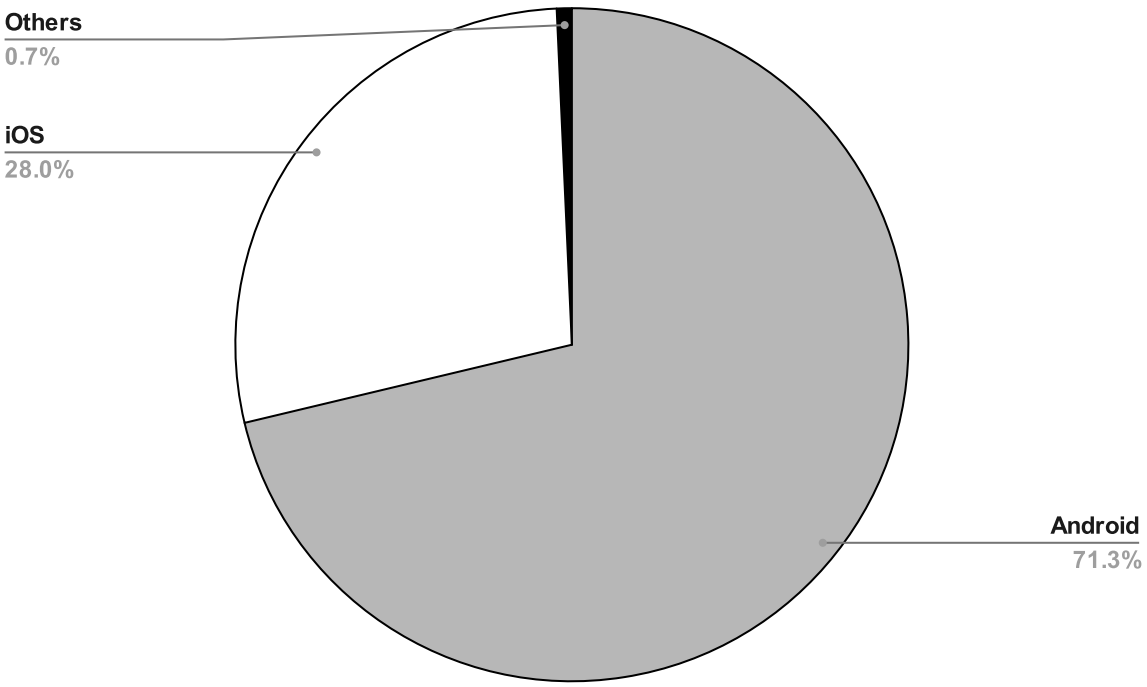


Figure 5 – Global mobile operating system market share in 2022 (extracted from STATCOUNTER (2022)).

ture. Nevertheless, we found some insightful tertiary studies in Software Engineering after a quick and informal literature search. For instance, Kitchenham et al. (2010) provided a tertiary study to analyze systematic literature reviews in Software Engineering. Hoda et al. (2017) discussed systematic literature reviews in agile software development, and Kudo, Bulcão-Neto e Vincenzi (2020) describes secondary literature focusing on requirement patterns.

This paper aims to present a comprehensive overview of current mobile application testing research and help researchers identify potential research topics for future work. In summary, this paper makes the following contribution:

- ❏ Summarization of 21 secondary studies related to mobile application testing.
- ❏ Categorization and overview of all selected studies according to the primary (General, Automation, Context-awareness, Environment) and specific research topics addressed by the studies.
- ❏ Characterization of the studies based on test objectives (functional and non-functional) and test platforms (Android and iOS).
- ❏ A discussion and analysis of 87 gaps and challenges identified by the selected studies.
- ❏ A research agenda with 15 open challenges for future work.

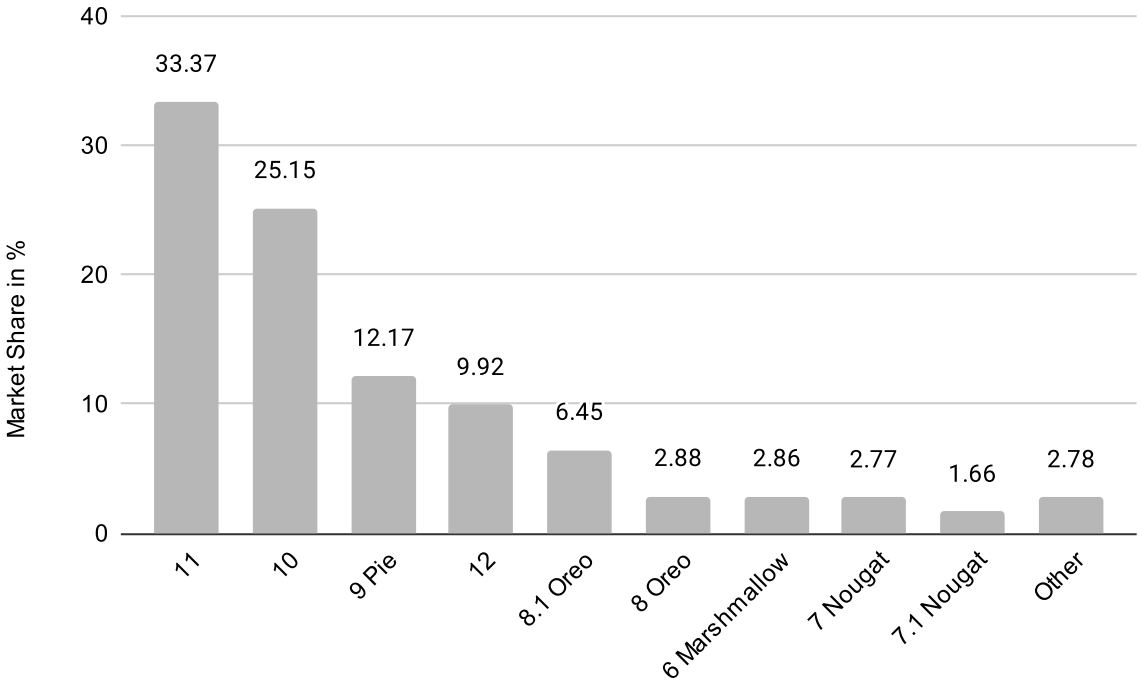


Figure 6 – Global mobile Android version market share in 2022 (extracted from StatCounter (2022)).

The remainder of this paper is organized as follows. Section 2.4 presents the study setup, including the research method, research questions, and the steps to carry out the study. Section 2.5 summarizes the results and answers the research questions. Section 2.6 discusses the results, provides a research agenda for future work, and presents the threats to validity. Finally, Section 2.7 concludes the paper.

2.4 Study Setup

2.4.1 Research Methodology

The present paper provides a comprehensive tertiary study in mobile application testing, presenting the trends, challenges, and gaps reported by the secondary studies available in the literature. Tertiary research follows the same procedure as a systematic literature review (SLR) (KITCHENHAM; CHARTERS, 2007). In general, a SLR encompasses three main phases (KITCHENHAM; CHARTERS, 2007; BRERETON et al., 2007; FABBRI et al., 2013):

– Planning: this phase consists of the definition of the goal, formalization of protocol (research questions, keywords, search strategy, sources, inclusion/exclusion criteria, and quality assessment criteria), and evaluation of the protocol.

– Execution: In this phase, the studies are identified according to the search strategy defined in the protocol. Next, the studies are selected conforming to the selection and quality criteria. At the end of the execution phase, the data are extracted and synthesized.

– Report: the results are evaluated and reported in this phase.

The following sections provide a complete description of the steps to conduct the tertiary study: definition of research questions, study selection process, data extraction protocol, quality assessment protocol, and a brief summarization of the results.

2.4.2 Research Question

Table 1 presents the research questions defined for this study.

Table 1 – Definition of research questions.

Research Questions	
RQ1.	*What secondary studies have been published in the field of mobile application testing?*
RQ1.1.	*What are the frequency and the types of secondary studies that have been published?*
RQ1.2.	*What is the quality of the secondary studies?*
RQ1.3.	*What are the primary studies addressed by the secondary studies?*
RQ2.	*What are the main and specific research topics addressed by the published studies?*
RQ2.1.	*What are the primary test objectives addressed by the published studies?*
RQ2.2.	*What are the primary test platforms considered by the published studies?*
RQ3.	*What are the main gaps identified by the secondary studies?*

The first research question provides a broad overview of included studies, showing the types (systematic literature review, systematic mapping study, literature surveys, or others), the frequency of studies per year, the quality of each study, and a brief analysis of primary studies covered by each secondary study. The second research question categorizes the studies according to the main and specific research topics. Moreover, this research question provides a comprehensive overview of the test objectives and platforms considered by the secondary studies. The third research question groups the main gaps and challenges identified by each secondary study and gives insights for future research.

2.4.3 Study selection process

The study selection process comprises four main steps presented in Figure 7.

2.4.3.1 Automatic search

Initially, we defined the online repositories to execute the search. The online databases chosen for the study are: *IEEE Xplore*, *ACM Digital Library*, *Scopus*, *SpringerLink*, *ScienceDirect*, *Engineering Village*, *ISI Web of Science*, and *Wiley*. It is important to emphasize that we considered those databases available at the university.

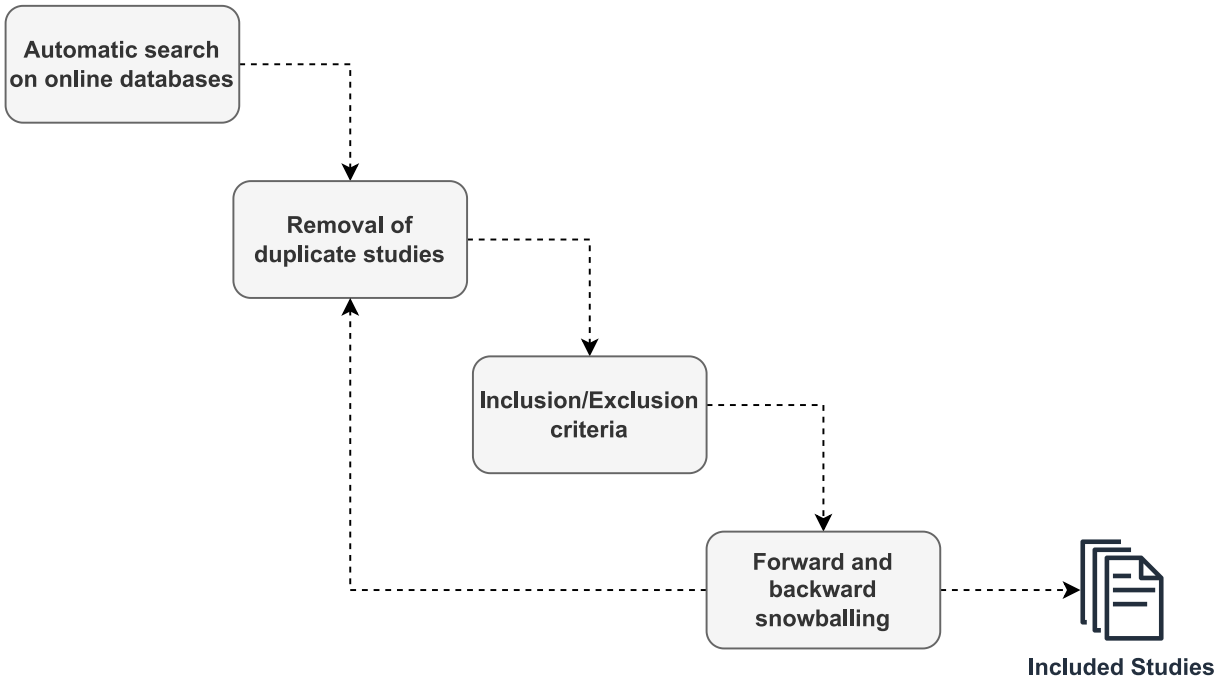


Figure 7 – Study selection process.

The next step consists of the formalization of the keywords to search. The study encompasses three major terms: **mobile application**, **testing**, and **secondary studies**. We also defined possible synonyms to enhance the search string:

- *mobile application*: mobile application, mobile app.
- *testing*: testing, test.
- *secondary studies*: systematic literature review, systematic review, systematic literature mapping, systematic mapping, mapping study, literature review, survey.

We combined the abovementioned keywords using the boolean connector **OR** to merge synonyms and the boolean connector **AND** to link major terms. Table 2 shows the search string applied in each attempt.

After defining the preliminary search string, we performed a pilot search (T1) on a single database (i.e.,*IEEE Xplore*) to validate and refine the search string. T1 resulted in a set of 108 papers. After reading the title, keywords, and abstract, there were three candidate papers for inclusion.

In the second attempt, we combined **mobile application** and **testing** to check whether the search would produce a more reliable set of papers. We observed that some studies focused on a specific operating system, and hence, we added **android** and **ios** as possible synonyms for mobile.

The second try retrieved 11 papers, and after analysis, we found a candidate paper not found by T1. Next, we evaluated T2 in a different database (i.e., *ACM Digital Library*). The search resulted in 137 papers with nine possible candidates for inclusion.

Table 2 – Search string attempt for pilot search.

ID	Search String
T1	(“mobile applications" OR “mobile application" OR “mobile apps" OR “mobile app") AND (“testing" OR “test") AND (“systematic review" OR “systematic literature review" OR “systematic mapping" OR “systematic literature mapping" OR “mapping study" OR “literature review" OR survey)
T2	(“mobile application testing" OR “mobile applications testing" OR “mobile app testing" OR “mobile apps testing" OR “android application testing" OR “android applications testing" OR “android app testing" OR “android apps testing" OR “ios application testing" OR “ios applications testing" OR “ios app testing" OR “ios apps testing") AND (“systematic review" OR “systematic literature review" OR “systematic mapping" OR “systematic literature mapping" OR “mapping study" OR “literature review" OR survey)
T3	(“mobile testing" OR “mobile application testing" OR “mobile applications testing" OR “mobile app testing" OR “mobile apps testing" OR “android application testing" OR “android applications testing" OR “android app testing" OR “android apps testing" OR “ios application testing" OR “ios applications testing" OR “ios app testing" OR "ios apps testing") AND (“systematic review" OR “systematic literature review" OR “systematic mapping" OR “systematic literature mapping" OR “mapping study" OR “literature review" OR survey)

Finally, we decided to check if a broader keyword would produce different results, and hence, we added **mobile testing** to the search string. In this case, the number of retrieved papers would be more significant than T2. The results of T3 for both *IEEE Xplore* and *ACM Digital Library* were 16 and 200 papers, respectively. Also, T3 brought some candidate papers that T1 and T2 could not find. Thus, we defined the following search string:

(mobile testing OR mobile application testing OR mobile app testing OR android testing OR iOS testing) AND (systematic literature review OR systematic review OR systematic mapping study OR systematic mapping OR literature review OR mapping study OR survey)

The final search string was adapted for each online database. A.1 illustrates the search string for each database. The automatic search was performed using the functionality “advanced search”, and we did not restrict the search by year. However, the search covered studies available until March 2023.

In some repositories (for instance, *ScienceDirect*, *Scopus*, *Web of Science*, and *Wiley*), we filtered the results to “Computer Science” since the preliminary search retrieved some studies not related to the field of mobile application testing (e.g., studies related to HIV testing).

Some online repositories limit the search of maximum keywords per query. For instance, *ScienceDirect* limits the search to a maximum of eight boolean connectors. In this case, we performed multiple searches with different permutations of the search string and merged the results into a single file, removing the duplicates ones.

We adopted StArt tool (FABBRI et al., 2016) to manage the references and to support the planning, execution, and summarization phases.

2.4.3.2 Removal of duplicate studies

In some cases, the same study appeared in different databases. Therefore, it was mandatory to remove the duplicate ones. StArt (FABBRI et al., 2016) automatically detects duplicated studies. However, there were cases that the tool was not able to remove. Thus, we manually analyzed the articles to remove the duplicated ones.

There might be cases where some studies were an extended version of a conference paper to a journal and presented the same idea and research methods. We analyzed the title, authors, year of publication, and abstract to decide whether the studies were duplicated, filtering out the least recent publication.

2.4.3.3 Inclusion and Exclusion Criteria

Next, we applied the inclusion/exclusion criteria to the remaining studies to filter out irrelevant ones and provide a more reliable set of papers (KITCHENHAM; CHARTERS, 2007). Table 3 describes the inclusion and exclusion criteria.

We relied on the title, keywords, and abstract to decide whether a study should be rejected. Whenever the paper's metadata was insufficient, we read the introduction and conclusion to decide. If a study met at least one of the exclusion criteria, it was marked as excluded. At the end of the process, if it was still unclear whether a paper should be excluded, we quickly read the full text to decide whether the study should be included.

Table 3 – Inclusion and exclusion criteria.

Inclusion criteria
(I1) Secondary study related to mobile application testing
Exclusion criteria
(E1) The study is not a secondary study
(E2) Study not available in full text
(E3) Study published as an abstract or a poster
(E4) Secondary study not related to mobile application testing
(E5) Study not written in English
(E6) Non-peer-reviewed study
(E7) Survey that is not a literature survey (for instance, questionnaire survey)

2.4.3.4 Snowballing

After selecting studies, we applied snowballing since a hybrid search (i.e., automatic search + snowballing) provides more relevant studies (MOURãO et al., 2020; FELIZARDO et al., 2016). This technique consists of checking the references of an article (i.e., *backward snowballing*) or the citations to a paper (i.e., *forward snowballing*) to identify additional relevant studies (WOHLIN, 2014). The present study considered both *backward snowballing* and *forward snowballing* to check whether the automatic search did not retrieve a secondary study.

Concerning *forward snowballing*, we used the "citation" option of *Google Scholar*[3] to collect all the references. It is important to emphasize that using *Google Scholar* to perform *forward snowballing* is sufficient to find the majority of studies not covered by the automatic search (Felizardo, Katia Romero et al., 2018).

2.4.3.5 Data Extraction

This section provides the protocol developed to extract all relevant information about the studies to conduct the research and answer the research questions. Table 4 presents the data extraction protocol.

Table 4 – Data extraction protocol.

Attribute	**Possible Value**
Title	
Author	
Year of Publication	
Type	SLR, SMS, Survey
# of Primary Studies	
Search Strategy	Automatic, Manual, Snowballing
Summary of Study	
Main Research Topic	
Speficic Research Topic	
Test Objectives	Functional, Non-Functional, Both
Test Platforms	Android, iOS, Other, Not Specified
Identified Gaps	

First, we collected the general information of each paper (i.e., title, author, year of publication, type of study, search strategy, and summary of study). These data gave us a broad overview and insights into the goal of each paper and helped answer RQ1, RQ1.1, RQ1.2, and RQ1.3.

Next, we collected the data for answers RQ2, RQ2.1, and RQ2.2. In this case, we were interested in each study's main and specific research topic, the concerned test objectives (functional, non-functional, or both), and whether the paper considered a particular test platform (Android, iOS, or not specified).

[3] https://scholar.google.com

To answer RQ3, we aggregated the gaps identified by each secondary study. These data provided a broad understanding of the open challenges and helped define a research agenda for future works.

2.4.3.6 Quality Assessment

The present study assessed the quality of the secondary studies using the same criteria applied by other tertiary studies (KUDO; BULCãO-NETO; VINCENZI, 2020). The Centre for Reviews and Dissemination (CDR) Database of Abstracts of Reviews of Effects (DARE)[4] of York University defined these criteria that comprise four questions to evaluate the quality of a given SLR. A score was assigned for each study from three possible values: Yes (1), Partially (0.5), and No (0). The final score varies from 0 to 4. Table 5 provides detailed information about the quality assessment protocol.

Table 5 – Quality assessment protocol adapted from DARE[4].

Q1. Were the inclusion/exclusion criteria properly described?
Y (1): Inclusion/Exclusion criteria explicitly defined
P (0.5): Inclusion/Exclusion implicit defined
N (0): Inclusion/Exclusion not defined and cannot be easily inferred
Q2. Did the search cover all relevant studies?
Y (1): Three or more relevant digital library and an additional search strategy
P (0.5): Three or more relevant digital library and did not include an additional search strategy
N (0): Two or fewer relevant digital library or the study did not show how the search was carried out
Q3. Did the authors assess the quality/validity of the included studies?
Y (1): Quality criteria were explicit and assigned to each primary study
P (0.5): Quality criteria were implicit or addressed by the research question of the study
N (0): The study did not present any quality assessment
Q4: Were the selected studies adequately described?
Y (1): Detailed information of each primary study was presented
P (0.5): Only a summary information of the studies was presented
N (0): The results of each individual study were not specified

2.4.4 Summary of the selection study process

The online search resulted in a total of 569 papers. Table 6 presents the number of papers retrieved per each online database.

The next step consists of removing duplicate studies and applying the inclusion and exclusion criteria to the retrieved papers. Figure 8 summarizes the results, showing the number of excluded and remaining articles on each step.

After the selection process, there were **eighteen** papers selected for snowballing phase to find any secondary study that was not retrieved by the automatic search. Concerning *backward snowballing*, we manually gathered the references of the eighteen articles into a

[4] To find the information regarding the quality protocol, access the URL: <https://www.crd.york.ac.uk/CRDWeb/AboutPage.asp>. Then, click on *About the databases*. Next, click on *About Dare*.

spreadsheet. We applied steps 2 and 3 (i.e., removal of duplicate studies and application of inclusion/exclusion criteria) to decide whether a paper should be added to the final set. Figure 9 presents the results of snowballing phase.

Table 6 – Number of studies retrieved per each database

Database	**# of studies**
ACM Digital Library	200
Springer	152
ScienceDirect	109
Scopus	29
Engineering Village	33
Web of Science	18
IEEE Xplore	16
Wiley	12
Total	**569**

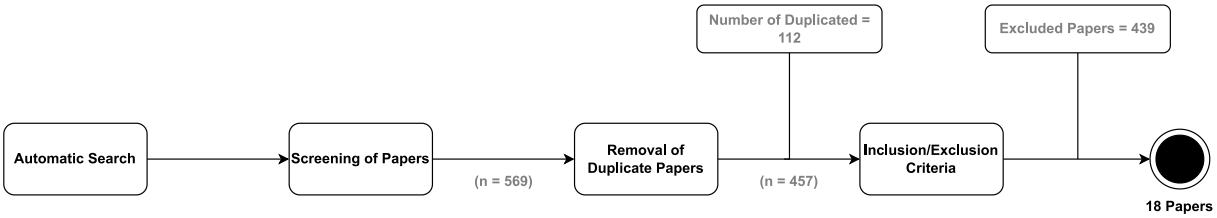


Figure 8 – Selection process steps include the number of duplicated, excluded, and included papers.

In the first round of snowballing, a total of 2.044 papers (1.508 studies from backward snowballing and 536 studies from forward snowballing) were manually analyzed. After the execution of steps 2 and 3, **three** new studies (two studies from *backward snowballing* and one study from *forward snowballing*) were added to the set of accepted papers. In the second round, we applied forward and backward snowballing to these three new articles, resulting in 1.002 studies for evaluation (114 studies from backward snowballing and 888 studies from forward snowballing). After analysis, no new studies have been selected, and hence, the final set encompasses **21 secondary studies**. A.2 presents the complete list of accepted papers. The studies S1 to S18 were retrieved from the automatic selection process, whereas S19 resulted from the first round of forward snowballing, and S20 and S21 resulted from the first round of backward snowballing.

The data of each secondary study have been extracted according to the protocol defined in Section 2.4.3.5. Then, we read the complete text, collecting all relevant information to answer the research questions. Finally, the quality of each study has been evaluated, conforming to the quality protocol presented in Section 2.4.3.6.

All the information regarding this tertiary study (e.g., protocols, the complete list of selected papers) is available online at <https://tinyurl.com/tertiary-study-repository>.

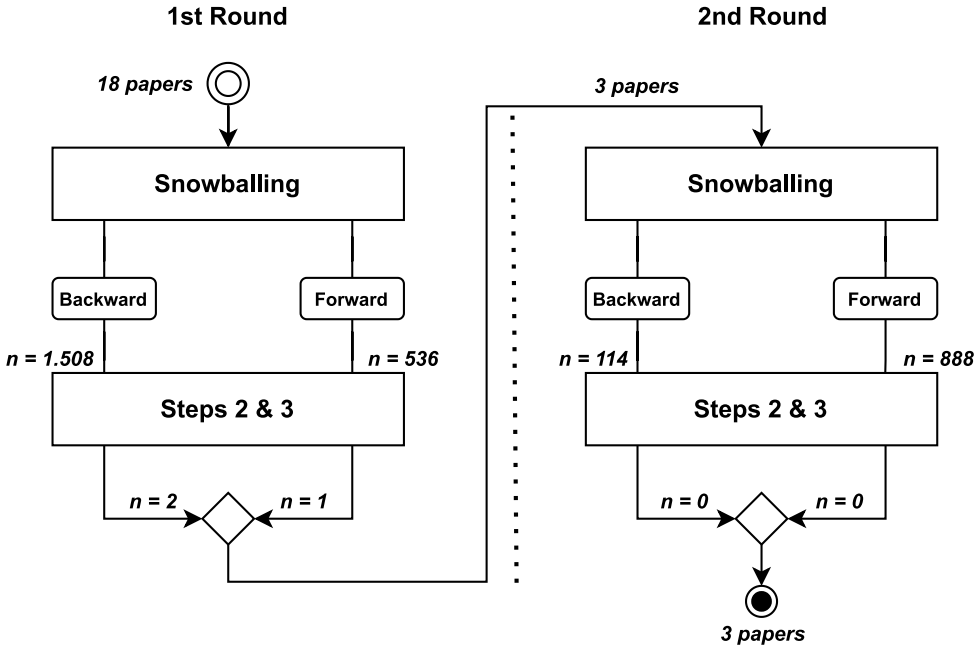


Figure 9 – Snowballing phases.

2.5 Results

The previous section described the research methodology applied in this study. The selection process resulted in 21 papers, and each article's data has been extracted and collected to answer the research questions. Table 7 summarizes the results from the tertiary studies, providing information on:

- ❏ Type of study: Systematic Mapping Study (*SMS*), Systematic Literature Review (*SLR*), Literature Survey (*S*), or Grey Literature (*GL*).
- ❏ Year of publication.
- ❏ Quality score of each study.
- ❏ Type of paper: Conference (*C*) or Journal (*J*).
- ❏ Research topic: General Aspects of Testing (*RT1*), Automation (*RT2*), Context-awareness (*RT3*), and Environment (*RT4*).

Section 2.5.1, Section 2.5.2, and Section 2.5.3 present the results for each research question.

Table 7 – Summary of the results of the tertiary study.

ID	Type of Study				Year of Publication	Quality Score	Type of Paper	# of Primary Studies	Research Topic			
	SMS	*SLR*	*S*	*GL*					*RT1*	*RT2*	*RT3*	*RT4*
S1	✓				2015	2	C	122	✓			
S2	✓	✓			2015	3	C	83		✓		
S3		✓		✓	2015	2.5	C	21	✓			
S4	✓				2016	1.5	C	230	✓			
S5	✓				2016	2.5	J	79	✓			
S6	✓				2017	3	C	51		✓		
S7		✓			2019	3	J	103		✓		
S8	✓				2019	2	J	68			✓	
S9	✓				2019	3	J	23				✓
S10			✓		2019	1	C	17	✓			
S11	✓				2019	3.5	J	131		✓		
S12	✓				2020	2.5	C	32	✓			
S13			✓		2020	1	J	52			✓	
S14		✓			2019	3.5	J	30				✓
S15	✓				2021	3	J	56	✓			
S16	✓				2021	3	J	16	✓			
S17	✓			✓	2022	2.5	J	50	✓			
S18	✓				2023	3.5	J	111	✓			
S19		✓			2020	2		19	✓			
S20			✓		2005	1	J	31	✓			
S21			✓		2013	1	C	17				✓

2.5.1 Answers to RQ1, RQ1.1, RQ1.2, RQ1.3 – Frequency, types, quality of secondary studies, and summary of primary studies

Figure 10 shows the number of secondary studies published by year. Until 2014, we found only two literature surveys (S20 and S21). In 2015, the number of studies started to increase until reaching its peak in 2019, with six secondary studies. The exception was in 2018, when we could not find any secondary studies published, considering our search criteria.

Concerning the type of study, Figure 11 shows most systematic mapping studies. On the other hand, the number of systematic literature reviews is practically the same as literature surveys. Moreover, one study presented systematic mapping and systematic literature review (S2), and two showed a multivocal literature review (S3 and S17). In S3, Kulesovs (2015) provides a systematic literature review and a grey literature review, whereas Villanes, Endo e Dias-Neto (2022) (S17) provides a systematic mapping study and a grey literature review.

To answer RQ1.2, we computed the quality score of each study according to the protocol defined in Section 2.4.3.6. Table 8 presents the results of the quality assessment.

As observed, none of the studies achieved a score of 4. Considering the quality attributes, Q1 and Q4 achieved the highest average quality score of 0.78. Therefore, we can assume that most secondary studies adequately described the inclusion/exclusion criteria

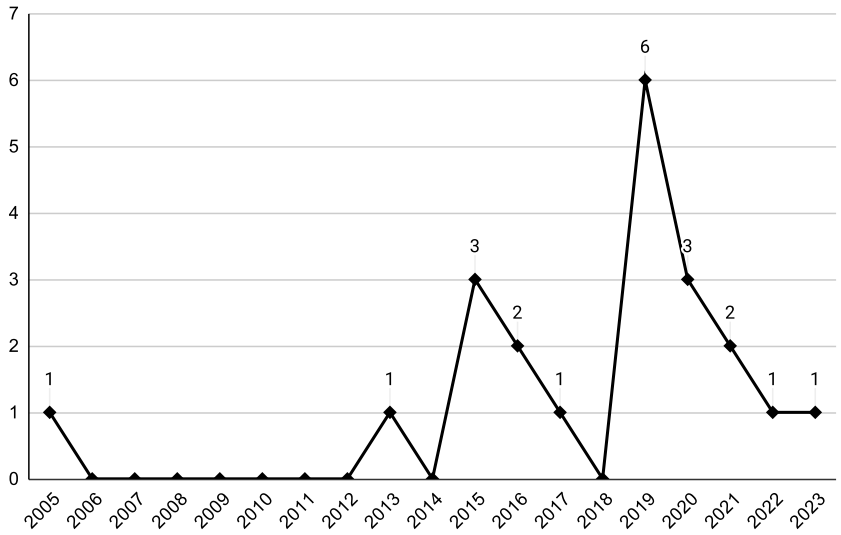


Figure 10 – Distribution of secondary studies per year.

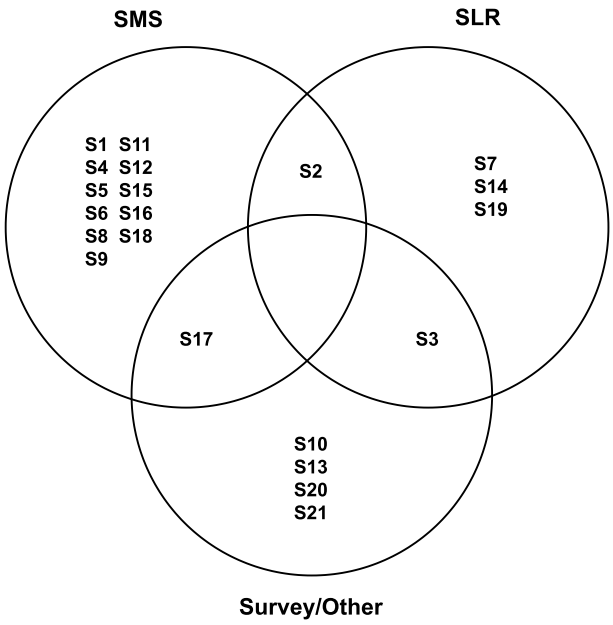


Figure 11 – Types of study.

and presented detailed information about the primary studies. On the other hand, Q3 has the lowest average score, i.e., most of the studies did not assess the quality of the primary studies. It is important to emphasize that quality assessment is not mandatory in systematic mapping studies (PETERSEN; VAKKALANKA; KURNIARZ, 2015). Since most studies are systematic mapping, it is expected that most of them did not provide a quality assessment of primary studies.

The highest quality score obtained was 3.5. Studies S11 and S14 did not consider an additional search strategy, whereas S18 did not provide detailed information about each primary study.

Table 8 – Quality assessment score assigned to each study.

ID	Q1	Q2	Q3	Q4	Total
S1	1	1	0	0	**2**
S2	1	0.5	1	0.5	**3**
S3	1	0.5	0.5	0.5	**2.5**
S4	1	0.5	0	0	**1.5**
S5	1	0.5	0	1	**2.5**
S6	1	0.5	1	0.5	**3**
S7	1	1	0	1	**3**
S8	1	0.5	0	0.5	**2**
S9	1	1	0	1	**3**
S10	0	0	0	1	**1**
S11	1	0.5	1	1	**3.5**
S12	1	0.5	0	1	**2.5**
S13	0	0	0	1	**1**
S14	1	0.5	1	1	**3.5**
S15	1	1	0	1	**3**
S16	1	1	0	1	**3**
S17	1	0	0.5	1	**2.5**
S18	1	1	1	0.5	**3.5**
S19	0.5	0.5	0	1	**2**
S20	0	0	0	1	**1**
S21	0	0	0	1	**1**
Avg	0.78	0.52	0.28	0.78	**2.38**
Std	0.40	0.36	0.43	0.33	**0.86**

Most studies did not satisfy one or more quality attributes, and the score was between 1.5 and 3. In this case, most of the papers considered just one type of search strategy (i.e., automatic search) or did not evaluate the quality of the studies.

The studies S10, S13, S20, and S21 obtained the lowest quality score (1). A reasonable explanation is that these studies were literature surveys and did not follow a rigorous method to conduct the research (e.g., the definition of a protocol, search strategy, and quality assessment).

Additionally, if we apply the same quality criteria to the present tertiary study, we obtain a score of 4.

2.5.1.1 Summary of primary studies

The present section provides a brief overview and statistics about the primary studies identified by the secondary studies. The 21 secondary studies resulted in 1.342 primary studies (see Table 9). All the information related to the primary studies was presented in the full text or the repositories created by the authors. There was one case (S2) where the link to the repository was inaccessible. However, we contacted the authors, who quickly shared all the needed information.

There were cases where the authors reported some inaccurate data. For instance, in S1, Sahinoglu, Incki e Aktas (2015) identified 123 primary studies related to mobile application verification. After analysis, we found a duplicate primary study on the final

list. In S13, Luo et al. (2020) reported 51 papers focusing on context simulation of context-aware mobile applications. However, we counted 52 primary studies. In S18, the authors reported 114 primary studies related to mobile GUI testing. However, we found three duplicated papers. It is important to emphasize that we collected the data from S18 based on the information in the full text presented by the authors. Besides, the repository did not contain complete information about the primary studies.

Table 9 – Number of primary studies identified by each secondary study.

ID	# PS	ID	# PS	ID	# PS
S1	122*	S8	68	S15	56
S2	83	S9	23	S16	16
S3	21	S10	17	S17	50
S4	230	S11	131	S18	111*
S5	79	S12	32	S19	19
S6	51	S13	52*	S20	31
S7	103	S14	30	S21	17
Total					1.342

* In S1, S13, and S18 there are some inconsistencies with the number of primary studies identified by the authors. In S1 there was a duplicated paper in the final list and in S13 we counted one study more than was reported. In S18 there were three duplicated papers.

The mapping study of Holl e Elberzhager (2016) (S4), focusing on the quality assurance of mobile applications, retrieved the largest number of primary studies. On the other hand, the mapping study of Silva et al. (2022) (S16) has returned the smallest number of primary studies (16), showing that interest in mutation testing for mobile applications is still growing.

There were cases where the same primary study has been cited by different secondary studies, as exemplified in Table 10. Ten different secondary studies cited both (AMALFITANO et al., 2013) and (AMALFITANO et al., 2015). For instance, the work of Amalfitano et al. (2013) was cited by S1, S2, S3, S4, S5, S7, S8, S11, S13, and S21. Despite the title suggesting that the study focuses on context-aware (such as S8 and S13), it has been cited in secondary studies that present general aspects of mobile application testing (S1, S3, S4, S5), automation (S2, S7, and S11), and cloud (S21).

Table 10 – Most cited primary studies.

Ref	Title	Authors	Cited By
(AMALFITANO et al., 2013)	Considering Context Events in Event-Based Testing of Mobile Applications	Amalfitano, D.; Fasolino, A.R.; Tramontana, P.; Amatucci, N.	S1, S2, S3, S4, S5, S7, S8, S11, S13, S21
(AMALFITANO et al., 2015)	MobiGUITAR: Automated Model-Based Testing of Mobile Apps	Amalfitano, D.; Fasolino, A.; Tramontana, P.; Ta, B.; Memon, A.	S1, S2, S4, S5, S6, S7, S8, S11, S18, S19
(LIU; GAO; LONG, 2010)	Adaptive random testing of mobile application	Liu, Z.; Gao, X.; Long, X.	S1, S2, S4, S5, S7, S11, S13, S14, S21
(AZIM; NEAMTIU, 2013)	Targeted and Depth-first Exploration for Systematic Testing of Android Apps	Azim, T. and Neamtiu, I.	S2, S4, S5, S6, S7, S8, S11, S18, S19

After applying a filter to remove the duplicate ones, we found that the 21 secondary studies aggregated 783 primary studies. Once again, all the relevant information regarding the primary studies can be found at <https://tinyurl.com/tertiary-study-repository>.

Figure 12 shows the frequency of primary studies per year. The first study was reported in 1995 by S20. After 1998, the number of studies started to grow until it reached its peak in 2015. However, after 2016, the number of primary studies rapidly decreased. Since the number of primary studies has reduced over the years, it is expected a reduction in the number of secondary studies reported (see Figure 10).

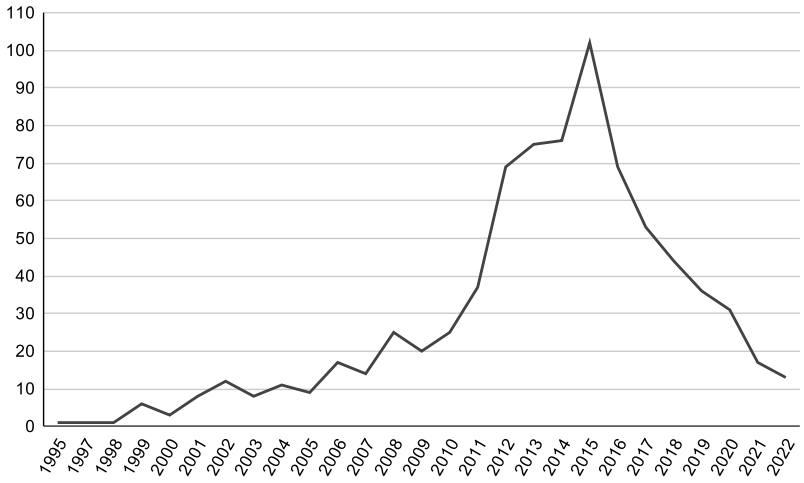


Figure 12 – Frequency of primary studies per year.

However, it is not possible to conclude whether this reduction in the number of primary and secondary studies occurred due to a lack of interest by the community or because there is still not enough evidence to propose new secondary studies. We performed an informal search on Google Trends[5] using four terms related to this tertiary study: mobile testing, mobile application testing, android testing, and iOS testing. The search was restricted starting in 2005 (the year of the first secondary study found) until 2023, and we use the option *Worldwide* for a broader search. Figure 13 presents the results.

When comparing Figure 13 and Figure 12, it is possible to note a correlation between them. In both cases, the number of searches/primary studies increased after 2009, reaching a peak in 2015. After 2015, the number of searches/primary studies slowly decreased. However, when assessing Figure 13, we observed that the searches for mobile testing, mobile application, Android testing, and iOS testing slightly increased in 2020, showing that these topics are still relevant.

Therefore, we expect that the results of this study provide insights to guide the proposition of future research.

[5] <https://trends.google.com/trends/explore?date=2005-01-01%202023-01-24&q=mobile%20testing,mobile%20application%20testing,android%20testing,iOS%20testing>

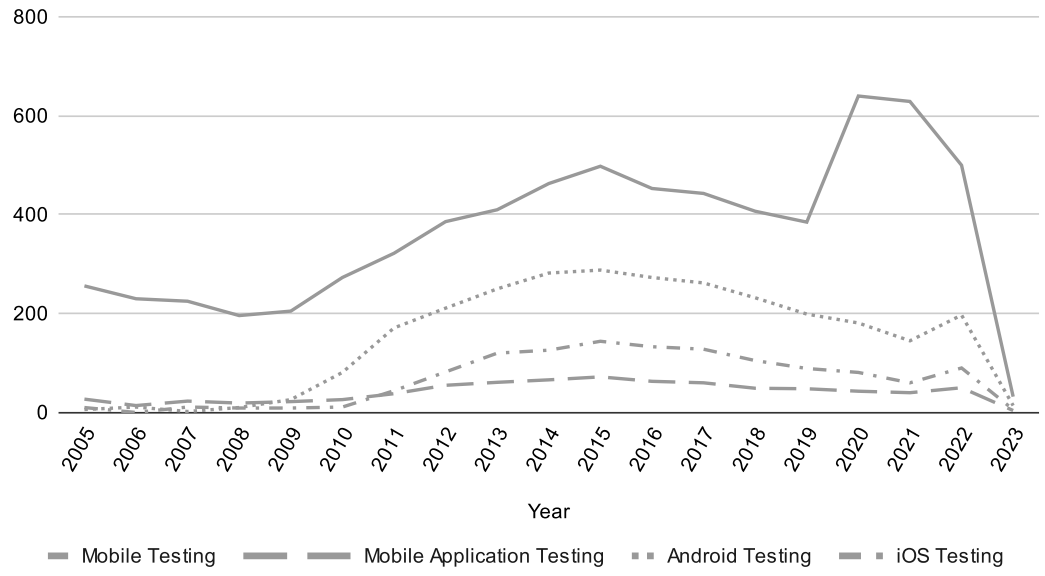


Figure 13 – Number of searches on Google regarding four terms: mobile testing, mobile application testing, android testing, and iOS testing.

2.5.2 Answers to RQ2, RQ2.1, and RQ2.2 – Main research topic, specific research topic, test objectives, and test platforms

2.5.2.1 Main research topics

First, we categorize the studies according to each study's main and specific research topics. This categorization was defined according to the purpose of the secondary study, and we collected the information during the data extraction phase (see Table 4). To derive the categories, we assessed each research topic addressed and identified the studies with common research topics. In this case, we defined a major classification scheme with four main groups: *General*, *Automation*, *Context-awareness*, and *Environment*. Next, we relied on the specific research topic addressed by each study to deduce the sub-categories. It is important to emphasize that a study may address a research topic related to another category. For example, automation may appear in studies categorized as General, but in this case, automation is not the main focus of the study. The description of the four main groups are:

- ❏ *General*: This category encompasses secondary studies that analyze methods, techniques, strategies or present a broad overview of a specific area;
- ❏ *Automation*: This category grouped studies that focus on automation in mobile application testing;
- ❏ *Context-awareness*: Context-aware applications are those applications that behave accordingly to context events, i.e., these types of applications use context informa-

tion as input to provide services and information to users (ALMEIDA; MACHADO; ANDRADE, 2020). This category groups secondary studies related to context-aware mobile applications;

- ❏ *Environment*: According to Gao et al. (2014), there are four types of infrastructure to perform testing activities: emulator-based, device-based, cloud-based, and crowd-based. This category encompasses studies focusing on a specific infrastructure or benefits from its usage based on these four types.

We identified the specific research topic addressed by the studies and grouped them according to their subgroups for each category defined above. Table 11 illustrates this categorization.

Table 11 – Categorization of the secondary studies per research topics.

Main Research Topic	**Specific Research Topic**	**Study ID**
General	*Overview*	S1 S3 S4 S5 S19
	Black / White Box	S10
	Energy Efficiency	S12
	Usability	S20
	NFR	S15
	Mutation Testing	S16
	Compatibility	S17
	GUI	S18
Automation	*Overview*	S2 S7
	Model-based	S6
	Functional	S11
Context-awareness	*Tools*	S8
	Context Simulation	S13
Environment	Cloud → *Overview*	S9 S21
	Cloud → *Security*	S14

General

In the first category, eight specific topics have been identified: *Overview*, *Black/White-Box*, *Energy Efficiency*, *Usability*, *GUI*, *Non-Functional Requirements (NFR)*, *Mutation Testing*, and *Compatibility*.

In *Overview*, the studies present a broad vision of a given subject area rather than focus on a specific topic. Thus, we identified five studies: S1 (2015), S3 (2015), S4 (2016), S5 (2016), and S19 (2020).

Concerning S1, Sahinoglu, Incki e Aktas (2015) conducted a systematic mapping study focused on the software verification of mobile applications. The paper relies on five research questions to identify the most used test types, the research issues addressed, the most frequently considered test level, and analyze the frequency of publications and journals to publish. As a result, the authors identified functional testing and usability as the most considered testing techniques and test environment management, test execution automation, and test case generation as the most addressed research issues. Moreover, the majority of studies focused on system-level testing. Finally, the authors identified challenges in performance testing and cloud usage in mobile application testing.

In S3, Kulesovs (2015) presented a secondary study combining a systematic literature review (SLR) and a multivocal literature review (MLR). The SLR aims to provide general aspects of mobile applications, whereas MLR seeks to identify aspects of iOS application testing. The studies from both reviews were classified according to characteristics (features and limitations) that influence iOS application testing: environment (hardware, operating system, resources, connectivity, internalization), application life-cycle, inside the application, and UI/UX. Next, the author discusses these characteristics regarding the literature review results based on professional experience. From S3, we argue that exists a lack of academic studies aiming at testing iOS applications.

Holl e Elberzhager (2016) (S4) provided a broad overview of the quality assurance of mobile applications. Overall, the research questions aim to identify: types of quality assurance approaches, testing level, phases and quality addressed, automation implemented, how the approaches are evaluated, and challenges. Similarly to S1, Holl e Elberzhager (2016) identified system-level testing and testing functionalities as the most considered test level and test objectives, respectively, considered by the primary studies. Also, most studies rely on frameworks and tools to implement automation. To conclude, the authors stated possible challenges: context simulation, security-by-design, device-specific test coverage, fault analysis for compatibility testing, user review for quality assurance, and test case generation regarding energy issues.

The systematic mapping study of Zein, Salleh e Grundy (2016) (S5) contributes to a comprehensive analysis of mobile application testing techniques. The authors investigate the techniques, approaches, contribution facets provided, applications used to assess the primary studies, and journals/conferences that include papers focusing on mobile application testing. The studies were categorized into five groups: usability testing, test automation, context-awareness testing, security testing, and testing in general. The results show that most studies focus on test automation, followed by usability testing (corroborating S1) and testing in general. Moreover, the authors provide an in-depth analysis of each study according to its category. Finally, some gaps were elicited, for instance, conducting research in real-world development environments and testing techniques for mobile service and life-cycle conformance.

In S19, Khan et al. (2020) also provided a secondary study (systematic literature review) focusing on testing techniques. However, the authors did not explicitly describe the research methods, protocol, search strategies, and inclusion/exclusion criteria. Also, there is no available link for a repository containing this information. Concerning the results, the authors summarized and discussed primary studies according to three groups: security and malware testing, cloud-based testing, and test automation.

Regarding *Black/White Box*, Hamza e Hammad (2019) (S10, 2019) summarizes black and white box approaches for testing web and mobile applications through a literature review. The authors discussed the strategies using four test-key factors: artificial intelligence, security, fully automated, and heuristic search. However, the study did not present guidelines on selecting the primary studies.

The systematic mapping presented by Moreira, Alves e Andrade (2020)(S12, 2020) focused on *Energy Efficiency* testing in Android applications. In S12, the studies have been categorized according to test generation, resource management inefficiency, fine-grained estimation, simulated run-time measurement, classification, and general. The results show that most primary studies were related to fine-grained estimation, followed by test generation, classification, and general. Next, the authors argued that most solutions were proposed as tools and frameworks, and most of the evaluation was carried out through empirical investigation or case study. In addition, most of the studies did not combine testing with another approach, and, in general, the technique relied on software solutions to measure power consumption.

For *Usability*, Zhang e Adipat (2005)(S20, 2005) presented a literature survey of usability testing in mobile application testing. This study aims to provide a literature review and define guidelines for usability testing in mobile applications. The authors discuss the literature according to methodologies, tools, usability attributes, and data collection methods. Then, they developed a framework for usability testing on mobile applications based on existing studies.

Júnior et al. (2021) (S15, 2021) proposed a systematic mapping study focusing on dynamic testing techniques of non-functional requirements (*NFR*). The study relied on twelve research questions to discuss the existing techniques and tools to support the testing of NFR. According to the results, performance, security, usability portability, reliability, and compatibility are the most concerned quality characteristics; Android is the most used testing platform, and black-box strategies are the most considered testing techniques. Moreover, Júnior et al. (2021) found that most of the non-functional requirements testing techniques are tool-supported, i.e., the existing tools address different testing of NFR, such as performance, usability, security, performance, and portability.

Silva et al. (2022) (S16, 2021) conducted a systematic mapping study on mutation testing of mobile applications. The authors defined seven research questions to find the trends, characteristics of primary studies, and evaluation aspects. Silva et al. (2022) high-

lighted the majority of studies had mutation testing as the primary goal, while few studies applied the technique as a form of assessment; most of the mutation operators were proposed to mutate source code in Java and XML; the majority of the tools only support mutant generation, and none of them support the three phases of mutation testing (mutant generation, mutant execution, and mutant analysis).

Villanes, Endo e Dias-Neto (2022) (S17, 2022) presented a multivocal literature mapping on mobile compatibility testing. According to the authors, most studies relied on hardware compatibility (i.e., different mobile devices and brands) rather than software compatibility (i.e., API and OS versions). Next, the authors presented a grey literature review to identify the most used criteria for mobile device selection. In this case, the authors identified sixteen characteristics used by practitioners for mobile device selection: OS version, OS, most used OS, most used OS version, screen size, new/upcoming devices, manufacturers, hardware, old devices, network, resolution, popularity, target audience, market share, devices for app goal, devices cause problems.

Finally, Nie et al. (2023)(S18, 2023) presented a systematic mapping study of GUI testing for mobile applications (*GUI* category). In S18, the primary studies were evaluated from two perspectives: (I) a bibliometric analysis to have a broad view of the mobile GUI testing area (e.g., research topics, influential papers, and authors); (ii) a qualitative analysis to understand the main research objectives, approaches, and evaluation metrics. Regarding the results, the authors found that test case generation and automated testing have gained much attention in mobile GUI testing. Moreover, most studies addressed testing functionalities of the application rather than non-functional requirements (reliability, performance, and security). Regarding techniques, model-based is the most used approach, followed by random-based and machine learning. To conclude, the authors found that code coverage and error detection are the most used metrics to evaluate the testing results.

Automation

Regarding automation, we identified *Overview*, *Model-based*, and *Functional* as specific research topics.

The *Overview* group encompasses studies that did not focus on a specific topic and provide a broad vision of a subject area. This group comprises two studies: S2 (2015) and S7 (2019).

In S2, Méndez-Porras, Quesada-López e Jenkins (2015) conducted a systematic mapping and review to investigate approaches, techniques, empirical assessment, and challenges in automated testing. The mapping study provided an overview of automation testing (i.e., journals/conferences, authors, and frequency of papers), while the review discusses the challenges, approaches, and evaluation methods. The results show that model-based testing is the most used technique for automation testing, and, in general,

the approaches were evaluated through case studies. Also, the authors argued that various context events, fragmentation, resource limitation, and changes in the mobile ecosystem are some challenges related to the automated testing of mobile applications.

The systematic literature review of Kong et al. (2019) (S7) analyzed primary studies related to automated testing of Android apps. The authors proposed a taxonomy of Android testing, which relies on four categories: Test Objectives, Test Targets, Test Levels, and Test Techniques. Each category was divided into subgroups. For instance, Test Objectives have been divided into Concurrency, Security, Performance, Energy, Compatibility, and Bug/Defect. Then, each primary study identified was grouped and discussed according to these subgroups. Some of the findings may corroborate the analysis of other studies. For instance, system testing (e.g., S1 and S4) and model-based testing (e.g., S2) have been widely used in automation. On the other hand, the authors argued a lack of studies focusing on compatibility testing and few approaches focusing on white-box and acceptance testing.

The second research topic identified is *Model-based* testing. The studies described before identified model-based testing as the most used technique for automated testing. The systematic mapping of Ascate et al. (2017)(S6, 2017) proposed to pinpoint the state-of-art of model-based testing techniques focusing on automation testing. In S6, the primary studies have been categorized according to the methods applied, research approaches, the model used to represent the structure of mobile applications, testing platforms, testing environments, and available tools. The results indicate that most studies use finite-state machines to represent their models, and most studies emulate their proposed approach considering Android. Despite model-based testing techniques being considered in many studies, the authors argued that few tools support novel MBT techniques, especially in the mobile context.

In the *Functional* group, the systematic mapping study of Tramontana et al. (2019) (S11, 2019) provided an in-depth analysis of automated functional testing of mobile applications. The authors defined four main goals, based on the GQM paradigm, to derive an extensive set of seventeen research questions. The goals include classifying, analyzing, and evaluating techniques and tools, categorizing addressed testing levels, and providing an overview of researchers, venues, and journals. Some of the findings reflect the results obtained by other secondary studies. For instance, most primary studies focused on model-based techniques for test case generation, and Android is the most considered test platform. To conclude, the authors discussed the need to reduce the gap between industry and academia and the absence of studies focusing on iOS application testing, context-aware testing, concurrency testing, and fault detection.

Context-awareness

This section discusses secondary studies related to testing mobile context-aware applications. Regarding specific research topics, we identified two groups: *Tools* and *Context Simulation*.

In *Tools*, Almeida, Machado e Andrade (2019)(S8, 2019) identified and discussed testing tools for Android context-aware applications. The authors analyzed the tools according to the implemented technique, highlighting those focusing on context-aware applications. Concerning the results, the authors found a total of 80 tools. However, only five tools are specific for testing context-aware Android applications, and another five general Android tools support context-aware applications. Similar to the works described above, S8 argued that most studies relied on MBT for test case generation and the difficulty of finding available tools. Besides, the authors argued that most of the tools implement GUI testing.

Concerning *Context Simulation*, Luo et al. (2020)(S13, 2020) surveyed the state-of-art of context simulation methods for testing mobile context-aware applications. The study did not present information on the research methodology, search strategy, or study selection. The surveyed studies were classified into two main approaches: data-driven and model-based. In addition, each approach comprises three phases: test case acquisition, test case refinement, and test execution. The authors provided an in-depth discussion of the primary studies according to their approach and phases. To conclude, Luo et al. (2020) summarize the main gaps: the need to apply context simulation in early-stage testing, improvements of emulators, handling heterogeneous contextual data, multi-device context-aware applications, and automation support.

Environment

This section describes secondary studies related to the testing environment. During the search, we could only find studies related to testing on the Cloud. Thus, we subcategorized those studies in *Overview* (i.e., studies that provide an overview of the subject area) and *Security* (i.e., studies focusing on security issues).

Regarding *Overview*, we found two studies: S9 (2019) and S21 (2013). In S21, Al-Ahmad, Aljunid e Sani (2013) provided a literature survey considering three aspects of testing: cloud testing (i.e., tests that use cloud resources to simulate real-world environments (JUN; MENG, 2011), mobile testing (i.e., testing activities for native and web applications on mobile devices (GAO et al., 2014)), and mobile cloud computing testing (i.e., testing applications where the data storage and processing are done in the cloud rather than mobile devices (DINH et al., 2013)). The study discussed these three aspects

based on current research. It argued the necessity to develop a model to test mobile cloud computing (MCC) applications due to their uniqueness and complexity. However, the authors did not describe the methodology to retrieve the primary studies.

In S9, Yaú et al. (2019) conducted a comprehensive systematic mapping study on cloud-based mobile application testing. The authors defined three research questions to identify the types of cloud-based mobile application testing, approaches and contribution facets provided by the empirical studies, and the effects of Testing-as-a-Service (TaaS) in mobile testing. The primary studies have been classified into six categories: security testing, compatibility testing, functional testing, GUI testing, and general. Regarding the results, the authors identified an increase in studies focusing on mobile application testing and a prevalence of studies considering Android as the test platform. Concerning the types of testing, most empirical studies focused on automated testing, followed by security testing. Many studies relied on frameworks to propose their approaches regarding the contribution facets. To conclude, the authors argued about the need for future research focusing on TaaS.

Concerning *Security*, Al-Ahmad et al. (2019)(S14, 2019) provided a systematic literature review focusing on penetration testing of MCC applications which aims at providing a security testing method to identify security assurance issues and application vulnerabilities (AL-AHMAD; ALJUNID; ISMAIL, 2020). In S14, the authors described the need to perform penetration testing, requirements, and available models for MCC. Next, the study discussed the testing requirements of MCC applications and identified the available models (e.g., penetration models of networks, penetration testing models targeting web vulnerabilities, and penetration testing models focusing on mobile applications). To conclude, Al-Ahmad et al. (2019) argued that there is a lack of studies aiming at modeling or addressing penetration testing on mobile cloud computing applications.

Finally, Table 12 presents a traceability matrix to summarize the research topics addressed by each study.

Table 12 – Traceability matrix summarizing research topics addressed by each study.

ID	General								Automation			Context awareness		Environment	
	O	*B/WB*	*EE*	*U*	*NFR*	*MT*	*C*	*GUI*	*O*	*MB*	*F*	*TO*	*CS*	*C→O*	*C→S*
S1	✓														
S2									✓						
S3	✓														
S4	✓														
S5	✓														
S6										✓					
S7									✓						
S8												✓			
S9														✓	
S10		✓													
S11											✓				
S12			✓										✓		
S13															
S14															✓
S15					✓										
S16						✓									
S17							✓								
S18								✓							
S19	✓														
S20				✓											
S21														✓	

2.5.2.2 Answers to RQ2.1 – Test Objectives

According to ISO/IEC/IEEE 29119-1 (ISO/IEC/IEEE 29119-1, 2022), test objectives refer to the rationale of performing testing, i.e., testing to validate the implementation and identification of defects and measure quality attributes. Regarding test objectives, we identified studies related to *functional testing* and *non-functional testing*. We also found studies that considered *both* test objectives.

From the Venn diagram illustrated in Figure 14, most secondary studies focused both on functional and non-functional testing (S1, S3, S4, S5, S7, S9, S18, S19, and S21). However, according to S1, S4, S7, and S18, most primary studies rely on testing functional requirements.

On the other hand, five studies focus on non-functional testing. For instance, S12 aimed at energy efficiency in Android apps; S14 addressed security issues through penetration testing; S15 highlighted dynamic testing techniques on non-functional requirements; S17 presented a multivocal literature mapping of compatibility testing; and S19 provided an overview of usability testing in mobile applications.

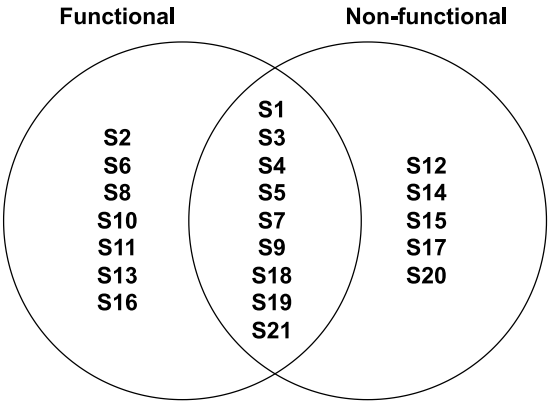


Figure 14 – Frequency of studies according to test objectives.

Table 13 provides a traceability matrix to show the non-functional requirements covered by the studies. In this case, we considered the quality attributes considered by the ISO/IEC 25010 Software Quality Model (ISO/IEC 25010, 2011). Observe that Security, Performance Efficiency, Compatibility, and Usability are the most considered quality attributes. On the other hand, few studies focus on portability, suitability, and reliability.

2.5.2.3 Answers to RQ2.2 – Test Platforms

Regarding test platforms, there are studies related to a specific platform (Android and iOS) and studies that did not address any platform. Figure 15 shows the distribution of studies per test platform.

Table 13 – Traceability matrix of non-functional requirements covered by the secondary studies according to the ISO/IEC 25010 Software Quality Model (ISO/IEC 25010, 2011).

ID	Software Product Quality							
	Suitability	Performance Efficiency	Compatibility	Usability	Reliability	Security	Maintainability	Portability
S1		✓	✓	✓				
S3	✓	✓	✓	✓			✓	✓
S4		✓	✓	✓	✓	✓		
S5				✓		✓		
S7		✓	✓			✓		
S9			✓			✓		
S12		✓						
S14						✓		
S15		✓	✓	✓	✓	✓		✓
S17			✓					
S18		✓			✓	✓		
S19						✓		
S20				✓				
S21		✓				✓		

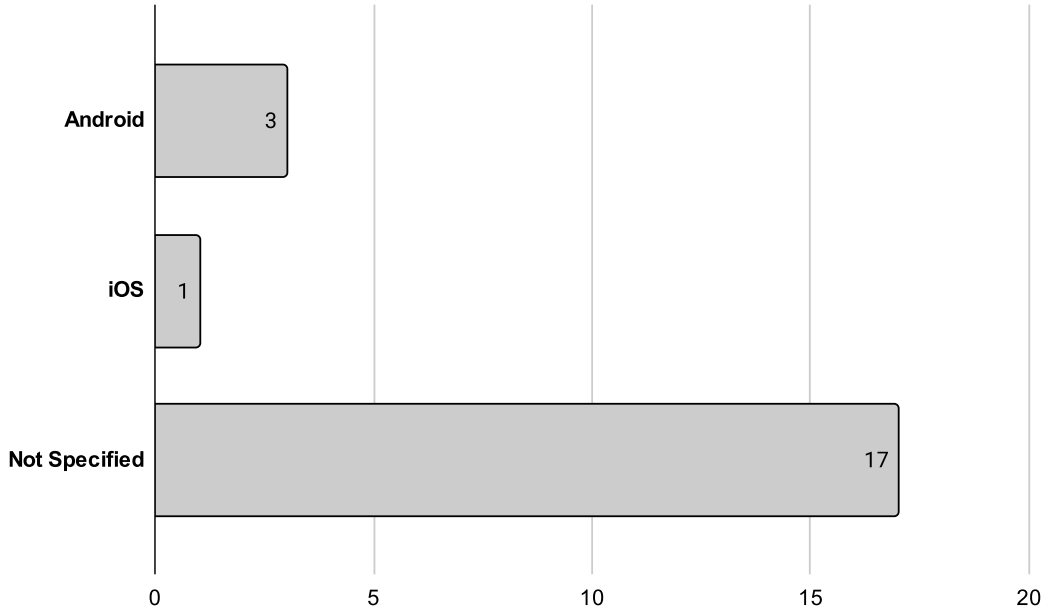


Figure 15 – Frequency of studies addressing a specific test platform.

The majority of studies did not directly address a test platform. However, in S5, S6, S9, S11, and S16, the authors explicitly identified Android as the most concerned test platform. For instance, in S16, Silva et al. (2022) pinpoint that all primary studies considered Android to perform their experiment.

Three studies considered Android as the main platform: in S7, Kong et al. (2019) provided an overview of automated testing of Android apps; in S8, Almeida, Machado e Andrade (2020) analyzed testing tools for Android context-aware applications; in S12, Moreira, Alves e Andrade (2020) mapped studies related to energy efficiency testing in Android apps. Moreover, we found one study related to iOS: in S3, Kulesovs (2015) addressed iOS application testing through a multivocal literature review.

2.5.3 Answers to RQ3 – Gaps and Challenges analysis

To answer RQ3, we first provide a Gaps and Challenges analysis based on the gaps and challenges identified by each secondary study. A.3 presents all identified gaps. The 21 secondary studies resulted in 87 research gaps and challenges. For each identified gap, we assigned an ID with the label **GCx.y**, where *x* refers to the study ID and y to the number of the identified gap of *x*. For instance, *GC1.3* stands for gap number three of study S1.

After analyzing the 87 research gaps and challenges, we found that some may appear in different studies. Hence, we categorized these 87 gaps and challenges based on their similarities, as presented in Table 14. It is worth noting that a given gap may appear in different categories. For instance, *GC8.1* states the need for tools to generate and execute test cases using high-level context. Hence, *GC8.1* was added to categories C1, C3, and C4.

Table 14 – Categorization of the identified gaps.

Cluster ID	Category	Gaps & Challenges ID	Studies ID
C1	Test Case Generation	GC2.1, GC4.6, GC8.1, GC12.2, GC12.3, GC16.2, GC18.2	S2, S4, S8, S12, S16, S18
C2	Non-functional Requirements	GC1.1, GC4.3, GC4.7, GC5.4, GC7.1, GC7.2, GC7.4, GC9.3, GC15.1, C15.2, GC15.3, GC15.5, GC17.1, GC17.2, GC17.3, GC18.5, GC20.1	S1, S4, S5, S7, S9, S15, S17, S18, S20
C3	Tools	GC2.4, GC4.8, GC6.5, GC7.3, GC8.1, GC11.3, GC15.3, GC16.6, GC19.1	S2, S4, S6, S7, S8, S11, S15, S16, S19
C4	Context-aware	GC4.1, GC8.1, GC8.2, GC11.5, GC13.1, GC13.2, GC13.3, GC13.4, GC13.5	S4, S8, S11, S13
C5	Cloud-related	GC1.2, GC2.3, GC9.1, GC9.2, GC9.3, GC9.4, GC9.5, GC14.1, GC14.2, GC14.3, GC21.1	S1, S2, S9, S14, S21
C6	Model-based Testing	GC1.3, GC2.2, GC6.1, GC6.2, GC6.3, GC6.4, GC6.5	S1, S2, S6
C7	Test Criteria	GC7.5, GC16.2, GC16.3, GC16.4, GC16.5, GC16.6, GC16.7, GC16.8, GC16.9	S7, S16
C8	Test Case Minimization and Prioritization	GC7.7, GC18.4	S7, S18
C9	Energy Efficiency	GC4.6, GC12.1, GC12.2, GC12.3, GC12.4, GC12.5, GC12.6, GC12.7, GC12.8	S4, S12
C10	iOS Testing	GC3.1, GC11.2	S3, S11
C11	Academia-Industry	GC4.7, GC5.1, GC11.1, GC15.4	S4, S5, S11, S15
C12	Journal/Venues	GC11.7, GC16.1	S11, S17
C13	Testing Techniques for Specific Purposes	GC4.5, GC5.3, GC7.6, GC11.4, GC18.3	S4, S5, S7, S11, S18
C14	Testing Process	GC5.2, GC18.1	S5, S18
C15	Test Environment	GC4.2	S4
C16	Fault Detection	GC11.6	S11

The first category (C1) groups the gaps and challenges related to *Test Case Generation*. In this case, six studies (S2, S4, S8, S12, S16, and S18) discussed the necessity of

investigating different facets of test case generation. For instance, *GC2.1* and *GC8.1* are gaps related to test case generation based on context, whereas *GC4.6* and *GC12.2* are gaps related to test case generation focusing on energy efficiency. In *GC18.2*, the authors argued about the need to define what constitutes the best test cases and how to generate the most effective ones. Since test case generation is a key part of the automation process, it would be interesting to assess how the area evolved through the years to understand the future directions for further research. Many techniques rely on MBT to generate test cases. However, with the advances in artificial intelligence, it would be interesting to investigate the usage of different techniques aiming at test case generation. For instance, the usage of a large language model (LLM) in the aforementioned context.

Category C2 encompasses the gaps and challenges related to non-functional requirements identified by eight studies (S1, S4, S5, S7, S9, S15, S17, S18, and S20). The gaps *GC1.1*, *GC4.3*, *GC4.7*, *GC5.4*, *GC7.2*, *GC7.4*, *GC9.3*, *GC18.5* highlight the need of studies aiming at specifics non-functional requirements (i.e., usability, performance, security, concurrency, and acceptance). On the other hand, the gaps and challenges identified by S15 (*GC15.1*, *GC15.2*, *GC15.4*, and *GC15.6*) focus on a general view of non-functional requirements for mobile apps. As observed in Table 13, there are some non-functional requirements attributes that require more attention from the research community: portability, maintainability, suitability, and reliability.

Category C3 grouped the gaps related to *Tools*. Ten different studies pinpointed the need to make tools available and scalable (*GC2.4*, *GC4.8*, *GC7.3*, and *GC15.4*), and tools to support a specific technique or framework (*G6.4*, *GC8.1*, *GC11.3*, *GC16.1*, and *G19.1*). It is worth noting that this issue may impact the proposition of new studies, especially those that rely on experimentation or those that compare a new approach with existing ones. For instance, AndroidRipper (AMALFITANO et al., 2012) is one of the most known tools in mobile application testing. However, when we assessed the repository of the tool, we observed that the tool had not been maintained for 6 years[6].

In C4, there are gaps and challenges related to context-awareness identified by four studies (S4, S8, S11, and S13). C5 grouped the gaps and challenges related to testing on the cloud identified by five different studies (S1, S2, S9, S14, and S21). These studies state the need to keep exploring the use of cloud infrastructure to support mobile application testing, the proposition of approaches to support multiple platforms and OS, and also conduct an investigation of different quality attributes when performing cloud-based mobile application testing (i.e., performance, usability, and security).

In C6, three studies (S1, S2, and S6) propose gaps and challenges focusing on Model-based Testing. This technique has been considered by different studies as the most used

[6] https://github.com/reverse-unina/AndroidRipper

technique related to automation. Hence, some topics may be worth further investigation, such as the validation of MBT techniques in real devices, the need to address fragmentation, and the provision of tools that support the existing MBT techniques.

Moreover, two studies (S7 and S16) present the need to explore different testing criteria for mobile application testing. In *GC7.5*, the authors stated the need for studies focusing on white-box approaches. In fact, only S10 provides an investigation of this topic. In S16, the authors state the need to continue exploring mutation testing in the context of mobile applications since few studies focus on this criteria.

Two studies (S7 and S18) identified gaps and challenges related to test case minimization and prioritization. For instance, in S7, the authors argued about using test case prioritization aiming at Android fragmentation. And finally, in C9, two studies(S4 and S12) present gaps related to energy efficiency.

The category C10 comprises gaps related to the need to explore iOS testing (*GC3.1* and *GC11.2*). In fact, there are few studies in the academic literature focusing on iOS testing. Many researchers tend to use Android OS due to its open-source nature. In C11 (*GC4.7, GC5.1, and GC11.1)*, the gaps and challenges reinforce the need to lessen the gap between academia and industry. It is worth noting that this issue can be applied to different areas of Software Engineering, i.e., this gap is not exclusive to mobile application testing. C12 encompasses gaps and challenges stating the need for specific venues for mobile application testing.

In C13, different studies (S4, S5, S7, S11, and S18) pinpointed the need to propose testing techniques for specific purposes. For instance, *GC5.3* focuses on techniques for life-cycle conformance and mobile services. *GC7.6* stated the need for regression testing techniques to identify defect-prone and unsafe updates. *GC11.4* argued the need for testing techniques focusing on C++ components of the Android NDK development framework.

In C14, the gaps focus on the enhancement of the existing testing process for GUI testing (*GC18.1*) and the need to elicit the testing requirements early during the development process *GC5.2*. Finally, C15 group a single gap that states the need for the testing environment (*GC4.2*), and C16 (*GC11.6)* group a single study related to fault detection. According to S11, few studies are focusing on bug localization and using historical information to find new bugs.

Figure 16 presents a timeline showing the number of secondary studies that identified a gap in a given category. This categorization showed that different studies recurrently address some gaps. In fact, both *Test Case Generation* (C1) and *Tools* (C3) have been constantly identified as a research gap, providing an interesting opportunity for future research. On the other hand, the number of studies related to Model-based Testing (C6) has decreased in the last few years.

Finally, we carefully analyzed whether a secondary study addressed an identified gap.

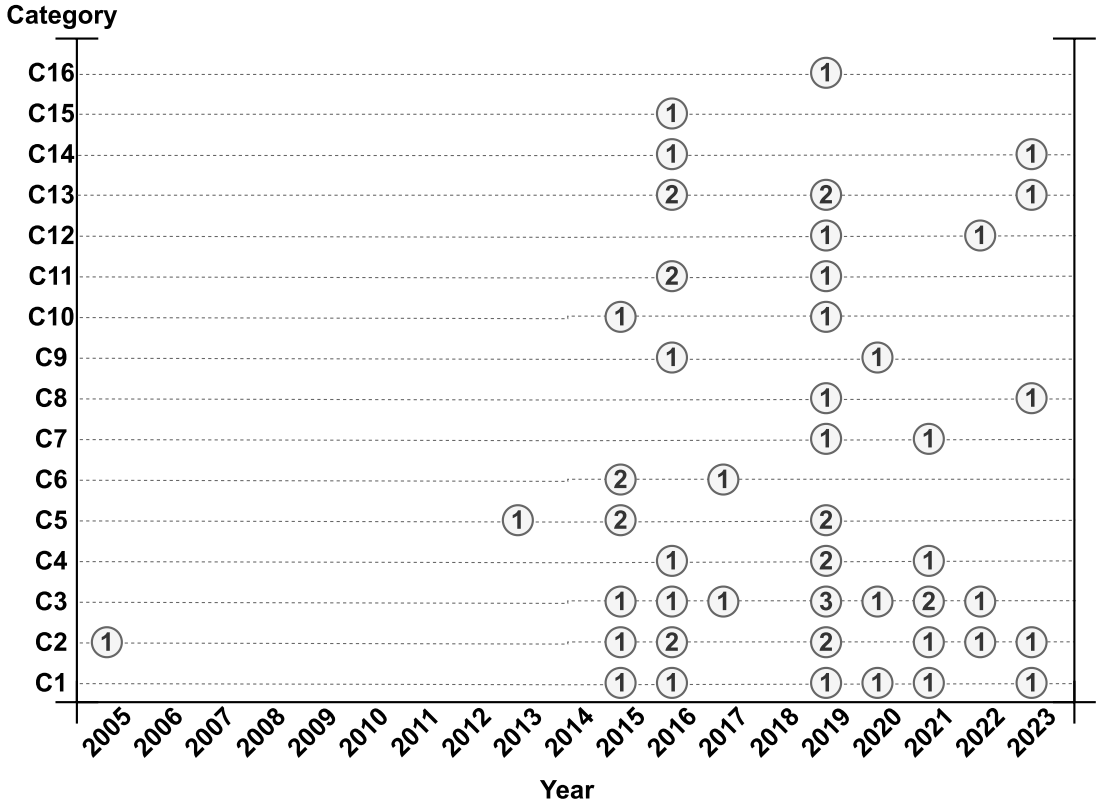


Figure 16 – Timeline of the proposed gaps and challenges and the number of secondary studies that addressed these gaps and challenges per year.

Figure 17 shows the gaps and challenges addressed by secondary research where the nodes represent the ID of secondary studies. The label on the edge represents which gap has been addressed. For instance, gap *GC21.1* stated the need for an MCC application testing model, and S14 discussed MCC application testing models, focusing on security issues. Hence, there is an edge from node S21 to S14, meaning that the gap *GC21.1* has been addressed by S14.

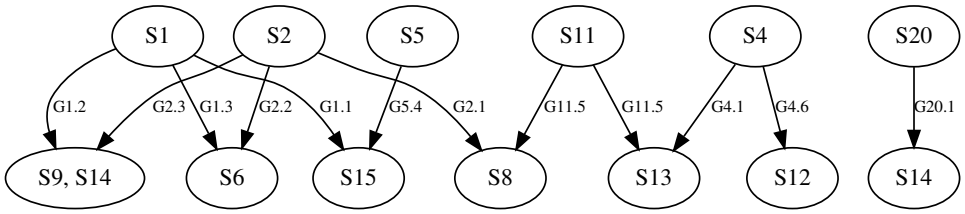


Figure 17 – Graph of gaps addressed by secondary studies.

There are cases where two different secondary studies have addressed the same gaps. In S1 and S2, the authors argued the necessity of testing on cloud (*GC1.2* and *GC2.3*, respectively). S9 presented a systematic mapping study on cloud-based mobile application testing, while S14 presented a systematic literature review on penetration testing of mobile cloud computing applications.

In S11, Tramontana et al. (2019) identified the need to conduct studies concerning context-awareness (*GC11.5*). We found two secondary studies related to context-awareness. In S8, the SMS of Almeida, Machado e Andrade (2019) discussed context-aware tools for Android applications, while S13 provided an overview of context simulation of mobile applications through a literature survey.

Still considering context-awareness, S2 discussed the need for test case generation based on context (*GC2.1*). S8 identified tools focusing on test case generation. In S4, Holl e Elberzhager (2016) stated the need for studies focusing on context simulation (*GC4.1*).

Two studies (S1 and S2) identified model-based testing as an open challenge (*GC1.3* and *GC2.2*). The SMS of Ascate et al. (2017) (S6) mapped primary studies on model-based testing for mobile applications.

Regarding non-functional requirements, studies S1 and S5 identified performance testing (*GC1.1*) and security and usability testing (*GC5.4*), respectively, as research gaps. In S15, Júnior et al. (2021) provides a systematic mapping study of non-functional requirements in mobile apps, which identified testing approaches for performance, security, and usability testing.

Finally, *GC4.6* states the need for research on test case generation concerning energy issues. The SMS of Moreira, Alves e Andrade (2020) (S12) identified primary studies focusing on test case generation regarding energy efficiency.

As presented in Figure 17, few secondary studies aim at structuring the state-of-art of a given research topic. In fact, there are still some topics that may be interesting to consider for future secondary studies. For instance, different secondary studies have cited test case generation as an open challenge.

Despite the study of Júnior et al. (2021) providing a systematic mapping of non-functional requirements testing techniques, few secondary studies address a specific NFR. Usability testing was pinpointed as one of the most considered NFR in mobile application testing (SAHINOGLU; INCKI; AKTAS, 2015; JúNIOR et al., 2021), and S20 is the only study directly addressing this research topic. However, S20 was published in 2005, and given the advances in the area and technologies, the study may be considered outdated. Hence, we suggest a proposition of an updated secondary study focusing on usability testing for future works.

2.6 Discussion

2.6.1 Discussion of the Results

Heretofore, we presented the results of our tertiary study, providing a comprehensive overview of mobile application testing. The research has been conducted based on three primary research questions.

We first identified the secondary studies published in academia. We observed a growth of secondary studies in the past few years, showing the maturity of the field (see Figure 10 on page 49). The results show that most studies present a systematic mapping to structure the research area. Furthermore, it is possible to observe many informal reviews that do not follow a well-defined protocol for conducting the studies. On the other hand, there are few systematic literature reviews, as presented in Table 15. Moreover, two SLRs (S2 and S7) focus on similar research topics, i.e., automated tests of mobile applications. For future research, we suggest more rigorous studies (i.e., systematic literature reviews) to analyze the topics that have not been covered or provide an update of published systematic mappings.

Table 15 – Existing SLR related to mobile application testing.

ID	Reference	Research Topic Addressed
S2	(MéNDEZ-PORRAS; QUESADA-LóPEZ; JENKINS, 2015)	Automated test of mobile applications
S3	(KULESOVS, 2015)	iOS application testing
S7	(KONG et al., 2019)	Automated test of Android applications
S14	(AL-AHMAD et al., 2019)	Penetration testing of mobile cloud computing applications
S19	(KHAN et al., 2020)	Testing techniques for smartphone applications

To answer the second research question, we categorized the studies according to four main groups: *General*, *Automation*, *Context-awareness*, *Environment*. The studies were grouped according to the specific research topic addressed (see Table 11). Most secondary studies aimed to provide an SMS focusing on a technique or an approach.

There are cases where different studies found the same results. For instance, three separate studies (S1, S4, and S7) identified functionality testing and system testing as the most concerned test objective and test level, respectively. Different studies (S2, S5, S7, S11, S18, and S19) identified model-based testing as the most concerned approach to automate the generation of test cases. Hence, we suggest future research studies on uncovered topics, such as different test case generation approaches.

Table 16 categorizes the studies concerning their research topic and test objectives. We observed a prevalence of studies focusing on functional testing, while only five secondary studies (S12, S14, S15, S17, and S20) directly addressed non-functional testing. From Table 16, we pinpoint that the automation of specific non-functional requirements and non-functional testing of context-aware applications are open challenges for future studies.

Regarding test platforms, most studies do not address a specific platform. However, most of them stated that Android is the most concerned platform. On the other hand, we found only one study (S3) concerning iOS. Therefore, we suggest the proposition of studies to explore testing on iOS for future research.

To answer RQ3, we provided a gap and challenge analysis to understand which challenges have already been addressed and which challenges are still open for future research. We collected 87 gaps and grouped them into ten categories (see Table 14 on Page 63). We found that test case generation and the lack of tools to support automation are two

Table 16 – Studies categorized according to their research topics and test objectives.

Categorization of secondary studies per research topics	Functional	Functional/ Non-Functional	Non-Functional
General → Overview		S1, S3, S4, S5, S19	
General → Black/White Box	S10		
General → Energy Efficiency			S12
General → Usability			S20
General → NFR			S15
General → Mutation Testing	S16		
General → Compatibility			S17
General → GUI		S18	
Automation → Overview	S2	S7	
Automation → Model-based	S6		
Automation → Functional	S11		
Context-awareness → Tools	S8		
Context-awareness → Context simulation	S13		
Environment → Cloud → Overview		S9, S21	
Environment → Cloud → Security			S14

major research gaps that secondary studies have systematically pinpointed. However, we could not find a secondary study that explicitly addresses the test case generation issue. It is worth noting that study S11 provides insights about this topic.

Once again, we pinpoint a proposition of secondary study to structure the topic of test case generation. Recently, different studies started investigating using the Large Language Model (LLM) to automatically generate test cases (LIU et al., 2023a).

Finally, eleven of these gaps were addressed by one or more secondary studies. To summarize the results, we present a word cloud of the 100 most frequently used words across 21 secondary studies. Figure 18 shows most of the findings.

For instance, a high frequency of the word “mapping” refers to many systematic mappings. Similarly, it is possible to note a prevalence of non-functional requirements related to security, usability, performance, energy, and compatibility. In addition, some gaps and challenges identified by each secondary study are displayed in the word cloud: test case generation and available tools.

2.6.2 Research Agenda

After analyzing 21 secondary studies, we identified some open challenges for future research regarding mobile application testing. As a contribution, we enumerated 15 topics to define a research agenda. These topics were defined according to the results of the proposed tertiary study. Moreover, we divided the topics into two categories: (i) Secondary studies challenges and (ii) Primary studies challenges. Concerning the first category, we list seven topics focusing on the proposition of potential secondary studies. In the second category, we pinpoint seven topics related to primary studies that could benefit the research community.

❑ Secondary studies challenges:

(i) Structure and analyze different techniques and tools for test case generation. Several studies mentioned test case generation as an open challenge and model-based testing as the most concerning technique to generate test cases. Additionally, many recent studies proposed different techniques for test case generation; for instance, AI-based approaches (COLLINS et al., 2021; KHAN; BRYCE, 2022).

(ii) Structure and analyze the state-of-art of testing tools and frameworks for mobile application testing. Many studies pinpointed the lack of available and scalable testing tools for different purposes.

(iii) Structure and analyze the state-of-art of security testing of mobile applications. Different studies mentioned security testing as the most concerned non-functional requirement testing. However, there is no secondary study directly addressing security testing.

(iv) Structure and analyze the state-of-art of testing context-aware mobile applications. There is a lack of secondary studies focusing on different techniques/environments to test context-aware applications and testing non-functional requirements of context-aware applications.

(v) Structure and analyze the state-of-art of using machine learning to support mobile application testing activities due to the increase in studies on using machine learning to support software testing activities (DURELLI et al., 2019).

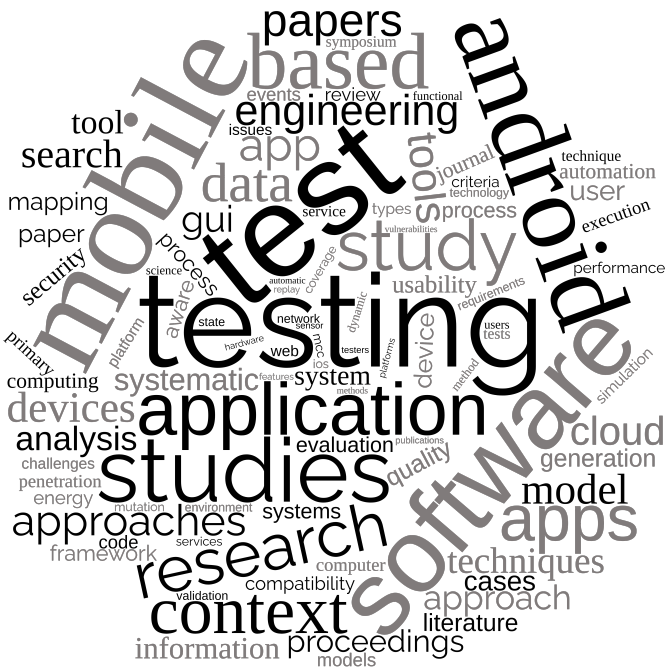


Figure 18 – Word cloud of the 100 most frequent words concerning the 21 secondary studies.

(vi) Structure and analyze the state-of-art of proposed infrastructures (based on cloud or to run locally) to support distributed tests on multiple real devices.

(vii) Structure and analyze the state-of-the-art multi-device and multi-platform testing aiming at fragmentation issues.

❑ Primary studies challenges:

(i) Investigate the design of frameworks for test case generation based on existing tools; for instance, Google Espresso[7] and Appium[8].

(ii) Investigate testing of different cross-platform frameworks such as React Native[9] and Flutter[10].

(iii) Investigate automated testing of non-functional requirements. As presented in Table 16, there is a lack of studies focusing on the automation of NFR.

(iv) Explore crowd-based testing in mobile application testing as an alternative for emulator-based, device-based, and cloud-based testing.

(v) Investigate approaches to optimize mobile application testing; the need to acquire different types of devices or the usage of existing infrastructure (AWS Device Farm[11], Firebase Test Lab[12]) to support testing may be costly for small companies.

(vi) Continue exploring different topics on mutation testing for mobile applications. For instance, equivalent mutant problem, mutant operator for Kotlin, and mutation testing for iOS, Flutter, and React Native.

(vii) Definition of a standardized benchmark to support experimentation since different studies rely on experimentation to validate their proposition.

(viii) The need to provide a standardized taxonomy for mobile application testing. We observed that most secondary studies propose their own taxonomy/classification scheme.

2.6.3 Threats to Validity

This section discusses the threats to validity. Several factors may affect the tertiary study, thus producing unreliable results. Below, we describe these threats and how we mitigate them.

[7] https://developer.android.com/training/testing/espresso
[8] https://appium.io/
[9] https://reactnative.dev/
[10] https://flutter.dev/
[11] https://docs.aws.amazon.com/devicefarm/
[12] https://firebase.google.com/docs/test-lab

Definition of protocol:

A tertiary study follows the same procedure as a systematic literature review. Therefore, this tertiary study relies on the guidelines proposed by Kitchenham e Charters (2007), which has been used in many secondary studies in Software Engineering. The protocol is a crucial factor of a systematic review since it specifies all relevant information of the study, such as research questions, type of search, online repositories, inclusion and exclusion criteria, quality assessment form, and data extraction form. In the present study, the first author defined the protocol, while the second and the third authors carefully reviewed it. We also provided an online repository containing all relevant information for cross-checking and replication.

Study selection process:

This threat concerns the risk of missing relevant studies. To mitigate this threat, we first conduct different searches to refine the keywords and search strings. We perform a hybrid search considering relevant online databases, both backward snowballing and forward snowballing, aiming to retrieve a precise set of relevant studies. Regarding study selection, the first author applied inclusion/exclusion criteria. If there was any uncertainty about whether a paper should be included, the authors discussed reducing any possible bias.

Presentation of the results:

This threat is related to a potential author bias in the data extraction process, influencing inaccurate results. For instance, although we have considered the quality form used in several other studies, the score assignment is subjective, depending on personal understanding. The first author executed the data extraction process, and the other authors carefully reviewed and criticized the obtained results.

2.7 Conclusion

Given the massive widespread of mobile devices, testing is an important activity for the quality assurance of an application and hence, provides the best experience for end-users. In the last years, the number of studies focusing on mobile applications has increased, showing the gaining of maturity of the area.

This study aims to provide a broad overview of mobile application testing by reviewing secondary studies. We found 21 secondary studies that address different research topics

related to mobile application testing. These articles have been categorized into four main groups: General, Automation, Context-awareness, and Environment. Then, we summarized the studies based on the specific research topic addressed by each study.

We also analyzed the studies based on test objectives and test platforms. Concerning test objectives, most papers focused on testing the functionalities of an application, whereas few studies directly addressed testing non-functional requirements. Regarding test platforms, most studies did not address any specific test platform. However, most empirical studies rely on Android to conduct their research, showing a lack of studies related to iOS application testing.

Finally, we identified all the gaps and challenges mentioned in each study and provided a gap and challenge analysis. This analysis categorizes the gaps and challenges based on their similarities and verifies whether a further secondary study has addressed a gap. This evaluation helped us evaluate which topics have been addressed by current research and which gaps still require broad investigation. After analyzing the results, we defined a research agenda including fifteen open challenges related to mobile application testing. We expect that our findings will be helpful in giving some insights and guiding further research in this area.

Chapter 3

Testing Infrastructures to Support Mobile Application Testing: A Systematic Mapping Study

3.1 Overview

This paper is the second paper in the timeline. It provides a systematic mapping study that structures and categorizes existing testing infrastructures to support mobile application testing. The decision to conduct a secondary study on this topic arose from the research agenda of the tertiary study presented in the previous chapter, which revealed a lack of studies characterizing the state of the art on this research topic. Moreover, this topic is aligned with the MAI/DAI project. The results helped us characterize the PhD research topic in fine detail. Below, the information regarding the paper is provided (see also Figure 19):

- **Title:** Testing infrastructures to support mobile application testing: A systematic mapping study
- **Authors:** Pedro Henrique Kuroishi (UFSCar), Ana Cristina Ramada Paiva (Universidade do Porto), José Carlos Maldonado (ICMC-USP), and Auri Marcelo Rizzo Vincenzi (UFSCar).
- **Local:** Information and Software Technology.
- **Year:** 2024.

❏ **Status:** Published.

❏ **DOI:** <https://doi.org/10.1016/j.infsof.2024.107573>

Figure 19 – Information regarding paper P2.

3.2 Abstract

Context: Testing activities are essential for the quality assurance of mobile applications under development. Despite its importance, some studies show that testing is not widely applied in mobile applications. Some characteristics of mobile devices and a varied market with different operating system versions lead to a highly fragmented mobile ecosystem. Thus, researchers put some effort into proposing different solutions to optimize mobile application testing. **Objective**: The main goal of this paper is to provide a categorization and classification of existing testing infrastructures to support mobile application testing. **Method**: To this aim, the study provides a Systematic Mapping Study of 27 existing primary studies. **Results**: We present a new classification and categorization of existing types of testing infrastructure, the types of supported devices and operating systems, whether the testing infrastructure is available for usage or experimentation, and supported testing types and applications. **Conclusion**: Our findings show a need for mobile testing infrastructures that support multiple phases of the testing process. Moreover, we showed a need for testing infrastructure for context-aware applications and support for both emulators and real devices. Finally, we pinpoint the need to make the research available to the community whenever possible.

3.3 Introduction

Nowadays, software is ubiquitous. The pervasive and ubiquitous characteristics of software products deeply impact society and change how people interact with each other and the environment. Moreover, the increasing digitalization of different individuals' and organizations' processes enhances the dependency on technological products. From this perspective, mobile devices and applications contribute to transforming this new paradigm of society (MUSHROOR; HAQUE; AMIR, 2019; SARWAR; SOOMRO, 2013; ISLAM; ISLAM; MAZUMDER, 2010).

In recent years, mobile devices have become increasingly popular. The report of Statista (STATISTA, 2023c) illustrates this scenario. In 2007, the number of smartphones sold worldwide was approximately 122 million. Ten years later, this number surpasses 1.5 billion. Moreover, the number of available mobile devices worldwide is expected to surpass 18 billion in 2025 (STATISTA, 2023a).

Different characteristics of mobile devices help to explain this gain in popularity. Mobile devices are portable and lightweight compared to traditional computers (e.g., desktops, laptops, and notebooks). These gadgets are equipped with different sensors and hardware components (e.g., GPS, camera, gyroscope, heart-rate sensors, wireless connectivity) that provide various services for end-users. Moreover, mobile devices allow the installation of mobile applications (or apps) from different domains (e.g., banking, entertainment, health, shopping), providing convenience in how people carry out various daily tasks. The widespread of mobile devices also impacts the mobile application market. In general, mobile applications are distributed through an app store. In December 2023, there were more than 2.4 billion available apps in Google Play Store, the official app store of the Android operating system (STATISTA, 2023b). Given this highly competitive business market, it became mandatory for IT companies to deliver high-quality apps for the end-users.

Software testing activities are essential for the quality assurance of a mobile application. Despite its importance, testing is still not widely adopted for mobile applications. The study of Pecorelli et al. (2021) shows that 60% of Android open-source apps have no test cases. Moreover, the quality of the test cases, when they exist, may be affected by possible test smells, i.e., poor design and implementation of test cases, which impact the quality of the test suite (DEURSEN et al., 2001; GAROUSI; KUCUK; FELDERER, 2019). Mobile application testing has some peculiarities that may increase its complexity (MUCCINI; FRANCESCO; ESPOSITO, 2012). As mentioned, mobile devices have different sensors and hardware components that may be considered when testing. Moreover, mobile devices may be designed with different screen sizes, densities, and versions of operating systems, depending on the manufacturer.

Observe that the wide variety of mobile devices provides a heterogeneous mobile ecosystem that causes a known issue named *fragmentation*. This phenomenon occurs

because manufacturers offer multiple mobile device models running different operating system versions. Note that this wide variety of mobile devices and OS may increase the complexity of testing activities due to compatibility issues for an application (LIU et al., 2023). Moreover, testing in all possible configurations of mobile devices is generally impracticable (WEI; LIU; CHEUNG, 2016). Hence, researchers put some effort into proposing strategies to reduce fragmentation impacts on mobile ecosystems (LIU, 2019; WEI; LIU; CHEUNG, 2016; LI et al., 2018; LANUI; CHIEW, 2019). Nowadays, commercial cloud services such as AWS Device Farm (AMAZON, 2024), Google Firebase Test Lab (LLC, 2024), and Visual Studio App Center (CORPORATION, 2024) offer a cloud infrastructure to run tests on various devices. These services provide a pay-as-you-go business model where the user pays according to the usage, i.e., per minute used or per test executed.

Recently, Kuroishi, Maldonado e Vincenzi (2024) presented a tertiary study assessing 21 secondary studies (i.e., Systematic Mapping Study, Systematic Literature Review, and Literature Survey) related to mobile application testing and proposed a research agenda with 15 open challenges. The results highlighted the need to provide an overview of existing testing infrastructure to support mobile application testing. Therefore, we present a Systematic Mapping Study (SMS) to categorize and classify existing studies that propose mobile application testing infrastructure.

In the context of mobile application testing, the study of Gao et al. (2014) describes four types of testing infrastructures:

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

For the systematic mapping study, we partially relied on the concepts presented by Gao et al. (2014) and envisioned a different classification for mobile testing infrastructures. In this case, we adopted and adapted the *cloud-based* and *crowd-based terminologies* and defined three types of testing infrastructures: **Cloud**, **Crowdsourcing**, and **Local**. Section 3.5.2.1 presents detailed information on this new proposed classification.

Overall, the present Systematic Mapping Study provides the following contributions:

- ❏ A new classification scheme for mobile application testing infrastructures.

- ❏ A categorization of 27 primary studies based on the types of testing infrastructure, types of supported operating systems, types of supported mobile devices, whether the proposed testing infrastructure is available for usage or experimentation, types of testing, and applications supported by the infrastructure.
- ❏ A description of existing research gaps to guide future studies.

The remainder of this paper is organized as follows: Section 3.4 presents the study setup, discussing all the steps required to perform the Systematic Mapping Study. Section 3.5 and Section 3.6 highlight and discuss the main finding of the investigation. Section 3.7 presents the threats to the validity and details how we mitigate them. Finally, Section 3.8 concludes the work.

3.4 Study Setup

3.4.1 Research Methodology

This paper aims to present a Systematic Mapping Study (SMS) to provide a broad overview of primary studies that propose or discuss testing infrastructures to support mobile application testing. To this aim, the study follows the guidelines proposed by Kitchenham e Charters (2007), Petersen et al. (2008), and Petersen, Vakkalanka e Kurniarz (2015).

The present Systematic Mapping Study has three main stages: planning, conducting, and presenting results. Figure 20 summarizes the tasks performed in this paper.

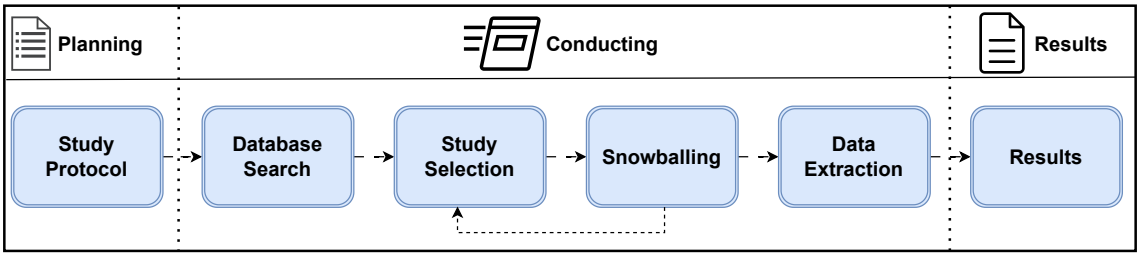


Figure 20 – Steps to carry out the systematic mapping study.

The study protocol was designed in the *Planning* stage to guide the SMS by defining the following activities: definition of the research questions (Section 3.4.2), the definition of the search strategy, the digital libraries, and the search string (Section 3.4.3), the definition of the data extraction protocol (Section 3.4.4) and definition of the quality assessment protocol (Section 3.4.5).

Concerning the second stage, i.e., *Conducting*, four tasks were performed (Section 3.4.6). The first task involves applying the search string in each digital library to collect the references. The second step consists of the selection process by defining a preliminary set of included studies which was used in the snowballing phase, aiming to find relevant studies not found by the database search. After defining the final set of included papers, all relevant information was collected based on the data extraction protocol aimed at answering the research questions. Detailed information about the Conducting stage is presented in Section 3.4.7.

The last step relies on presenting the main findings of the Systematic Mapping Study, where each research question was answered, pinpointing potential research gaps, and discussing the study's main contributions.

3.4.2 Research Questions

For the present study, two primary research questions to guide the Systematic Mapping Study. These two main questions led us to derive five fine-grained research questions as presented below.

1. **What studies focus on proposing testing infrastructure to support mobile application testing?**
    a) *What is the year of publication, the authors, and the venues of the primary studies?*
    b) *What is the quality of the included primary studies?*
2. **What infrastructures for mobile application testing are proposed?**
    a) *What are the types of proposed testing infrastructures?*
    b) *What are the types of operating systems (OS), devices, testing, and applications supported by the testing infrastructures?*
    c) *Is the proposed testing infrastructure available for usage or experimentation?*

3.4.3 Search strategy, the digital libraries, and the search string

Two search strategies were adopted for this SMS: database search on digital libraries and snowballing. It is known that a hybrid search that combines a database search in digital libraries and snowballing is efficient in finding relevant primary studies (MOURãO et al., 2020). The first strategy consists of running the search string in each digital library and collecting all the references. We adopted the StArt tool (FABBRI et al., 2016) to manage the references and to assist in the study selection process. Section 3.4.7 presents detailed information about the snowballing strategy adopted in this study.

For the database search, the following digital libraries were considered: **IEEE Xplore**, **ACM Digital Library**, **Scopus**, **Engineering Village**, **ScienceDirect**, **Web of Science**, and **Wiley**. These are well-known digital libraries that index Computer Science studies and were also used in different systematic mapping studies and systematic literature reviews (TRAMONTANA et al., 2019; KONG et al., 2019; ZEIN; SALLEH; GRUNDY, 2016).

Next, the search string for the digital libraries was defined. This process started by defining the major terms to compose the search string. For this study, three major terms were defined: **mobile**, **application**, and **testing infrastructure**. The next step consisted of defining possible synonyms for each major term as presented in Table 17.

Table 17 – Major terms and synonyms used to generate the search string.

Major Terms	**Possible synonymous**
mobile	Android
application	app
testing/test infrastructure	testing/test platform testing/test environment testing/test infrastructure-as-a-service testing/test infrastructure as a service testing/test service device lab device cloud mobile farm device farm testbed/test bed cloud service testing system

Concerning *mobile*, the term Android was added since this Operating System (OS) is the most considered in academic research related to mobile application testing. We tried to provide multiple synonyms for testing infrastructure to enhance the search for primary studies. However, we knew some terms could provide many studies unrelated to the SMS's goal. For instance, the term testing service provided many studies focusing on testing the component service of Android API[1] rather than proposing a testing infrastructure. In this case, the trade-offs were assessed and the term was maintained as possibly synonymous. All synonyms were merged using the boolean connector **OR** and the major terms using the boolean connector **AND**. The final search string is presented in Figure 21.

Observe that some synonyms may also be applied to the search string. For instance, for mobile, the terms "iOS", "Windows Phone", "smartphone", and "device" could be added to the search string. Additionally, "software" was a candidate for application. In this case, all these synonyms were added to the original search string (see Figure 21) and applied in the digital libraries. The results showed an increase in the number of retrieved

[1] https://developer.android.com/reference/android/app/Service

(mobile **OR** android)
AND
(app **OR** application)
AND
(test infrastructure **OR** testing infrastructure **OR** test platform **OR** testing platform **OR** test environment **OR** testing environment OR test infrastructure-as-a-service **OR** testing infrastructure-as-a-service **OR** test infrastructure as a service **OR** test infrastructure as a service **OR** test service **OR** testing service **OR** device lab **OR** device cloud **OR** mobile farm **OR** device farm **OR** testbed **OR** test bed **OR** cloud service **OR** testing system)

Figure 21 – Defined search string.

studies (over 32,000 studies). Additionally, this new search string did not bring any new candidate's papers for inclusion. Hence, we decided to maintain the original search string presented in Figure 21.

Finally, the search string for each database was adapted. It is worth noting that the string may vary for each database since they do not have a standardized search mechanism. Moreover, the ScienceDirect digital library is limited to 8 major terms per query. In this case, the search string was adapted by choosing the terms that could bring more relevant studies.

The database search considered studies indexed until September 2023. After collecting, extracting, and mapping all the relevant results, the authors performed an additional database search aiming at verifying if any new study was published during this period. In this second round, the search string was reapplied to the digital libraries considering the period from September 2023 to February 2024.

3.4.4 Data Extraction Protocol

Table 18 presents the data extraction protocol defined for the study. The protocol provides all the needed information to answer the research questions.

Table 18 – Data Extraction Protocol

Attribute	**Value**	**RQ**
Title		1
Author		
Year of Publication		
Venue	{Journal, Conference, Workshop, Symposium}	
Type of Infrastructure	{Local, Cloud, Crowdsourcing}	2
OS Supported	[Android, iOS, Windows Phone, N/A]	
Types of Device	[Real Device, Emulator, N/A]	
Available for Usage/Experimentation	{Yes, No}	
Types of Testing		
Types of Application		

The first column presents the attributes collected for each paper. The Value column corresponds to the possible values of the defined attributes. The attribute's value was considered as input, i.e., text or number, in cases where the rows were empty. The attributes Venue, Type of Infrastructure, OS Supported, Types of Device, and Availability for Usage/Experimentation have predefined values. In cases where the values are between braces, only a single value is chosen. For instance, the attribute Available for Usage/Experimentation can only be set with values Yes or No.

On the other hand, when the value is between square brackets, it is possible to choose multiple values. For instance, the infrastructure may support real devices and emulators. The value N/A means that the attribute was not explicitly presented.

Finally, the column RQ corresponds to which research question the attribute is associated.

3.4.5 Quality Assessment Protocol

The quality of the primary studies was evaluated to comprehend the maturity of the studies regarding the scientific rigor and relevance considering both the academic and industry point-of-view. It is worth noting that quality assessment is not mandatory in Systematic Mapping Study (PETERSEN et al., 2008).

Therefore, two different protocols for quality assessment were considered. The first protocol (PA) was defined by Dybå e Dingsøyr (2008), and we used it to assess the primary studies in terms of academic relevance since it is the most widely adopted quality assessment protocol (YANG et al., 2021). The second protocol (PI) was defined by Ivarsson e Gorschek (2011) and was used to assess the primary studies according to their industry relevance since it has been widely adopted for assessing studies with an industrial context (YANG et al., 2021). It is worth noting that both protocols have been applied in different secondary studies (YANG et al., 2021). Table 19 presents PA, and Table 20 presents PI.

The PA protocol has two possible values for each attribute: Yes (1) or No (0). When a given attribute was partially available, it was marked with the value Yes (1). The sum of each attribute gives the final score of this study, and a given study can have a maximum score of 11.

The primary studies are assessed in PI protocol by its industrial *Rigor* and *Relevance*. The score of *Rigor* is given by the sum of *Context (C)*, *Design (D)*, and *Validity threats (V)*. The score of relevance is given by the sum of *User/Subject (U)*, *Context (C)*, *Scale (S)*, and *Research Method (RM)*. The final score is given by the sum of *Rigor* and *Relevance*; hence, a primary study has a maximum score of 7.

PA and PI scores were standardized to a common base for a fair comparison since both scores were not the same dimensions.

Table 19 – Quality assessment protocol adapted from Dybå e Dingsøyr (2008).

Quality Assessment Protocol – Academia
QCA1. Is this a research paper?
QCA2. Is there a clear statement of the aims of the research?
QCA3. Is there an adequate description of the context in which the research was carried out?
QCA4. Was the research design appropriate to address the aims of the research?
QCA5. Are the subject cases defined and described precisely?
QCA6. Was there a control group, if any, to compare treatments?
QCA7. Was the data collected in a way that addressed the research issue?
QCA8. Was the data analysis sufficiently rigorous?
QCA9. Has the researcher identified potential threats to the validity of the study?
QCA10. Is there a clear statement of findings?
QCA11. Is the study of value for research or practice?

Table 20 – Quality assessment protocol adapted from Ivarsson e Gorschek (2011).

Rigor		
Context	*Strong Description (1)*	The context is described to the degree where a reader can understand and compare it to another context
	Medium Description (0.5):	The context in which the study is performed is briefly presented/mentioned in a degree that the reader cannot understand and compare it to another context
	Weak Description (0):	There is no description of the context in which the study is performed
Design	*Strong Description (1)*	The study design is described in a level where the reader understand the procedure to carry out the study, that is, the variables measured, the number of subjects, the sampling, treatments
	Medium Description (0.5):	The study design briefly described the factors related to design and data collection
	Weak Description (0):	The study design is not described
Validity Threats	*Strong Description (1)*	The threats to validity (i.e., internal, external, conclusion, and construct validity) are discussed and detailed
	Medium Description (0.5):	The threats to validity are mentioned but not described in detail
	Weak Description (0):	The threats to validity are not discussed/presented
Relevance		
User/Subject	*Contribution to relevance (1):*	The subjects used in the evaluation are representative of the intended users of the approach
	No contribution to relevance(0):	The subject used in the evaluation are not representative of the intended users of the approach
Context	*Contribution to relevance (1):*	The evaluation is performed in a setting representative of the intended usage, i.e., industrial setting
	No contribution to relevance(0):	The evaluation is performed in a laboratory setting or other settings not representative for real usage situation
Scale	*Contribution to relevance (1):*	The scale of the applications used in the evaluation is of realistic size, i.e., the applications are of industrial scale
	No contribution to relevance(0):	The evaluation is performed using applications of unrealistic size, i.e., toy projects
Research Method	*Contribution to relevance (1):*	The research method used in the evaluation is one that facilitates the investigation of real user situations (i.e., action research, case study, field study, interviews, descriptive/exploratory surveys)
	No contribution to relevance(0):	The research method used in the evaluation does not lend itself to investigate real user situations (i.e., laboratory experiment, conceptual analysis)

3.4.6 Study Selection Process

The study selection process consists of three steps. The first step consists of removing duplicated studies. In this case, a study is considered duplicated if it appears in different

databases or is an extended version of a prior study. For the latter case, the metadata (i.e., title, authors, year of publication, and abstract) was assessed to decide if the study was an extended version. Then, the least recent publication was marked as duplicated.

The second step involves applying the inclusion and exclusion criteria (IC/EC) to filter the most relevant studies. The IC/EC was applied based on each paper's title, abstract, and keywords. Whenever this information was insufficient, the authors read the introduction and conclusion to decide whether a paper should be included. For this study, the following inclusion criteria were defined:

- ❏ (IC1) A study that defines/presents an infrastructure/environment for mobile application testing.

Moreover, five exclusion criteria were defined:

- ❏ (EC1) A study that does not propose an infrastructure/environment for mobile application testing.
- ❏ (EC2) A study not written in English.
- ❏ (EC3) A study not available online.
- ❏ (EC4) A study published as an abstract, poster, or book chapter.
- ❏ (EC5) Non-peer-reviewed study.

In the third step, the authors read the full paper to decide whether a study should be included. At the end of this process, a preliminary set of included papers for the snowballing phase was defined.

3.4.7 Snowballing

In the next step, snowballing was performed in the preliminary set of papers. Snowballing is a technique to find new candidate studies for inclusion based on the references of a paper (i.e., backward snowballing) or based on the citation of the paper (i.e., forward snowballing) (WOHLIN, 2014). In this SMS, both backward and forward snowballing were considered.

For backward snowballing (BS), the references of each paper were collected into a spreadsheet. For forward snowballing (FS), the Google Scholar citation option was adopted to collect the references into a spreadsheet. It is worth noting that Google Scholar efficiently finds studies not covered by the database search (MOURãO et al., 2020). After collecting the candidate papers, the steps defined in Section 3.4.1 were applied, i.e., removal of duplicate studies, IC/EC criteria, and full-text reading. Figure 20

shows snowballing is an iterative task. Therefore, it is performed while new papers are found in each snowballing phase. The process ended since no new study for inclusion could be found.

3.4.8 Summarizing the Results of the Study Selection Process

Table 21 presents the number of studies of each digital library. Engineering Village and Scopus retrieved the largest number of studies (3,976 and 3,965, respectively). On the other hand, ScienceDirect retrieved 145 studies. One plausible explanation for this difference in the number of retrieved studies relies on the need to adapt the search string due to the limitation of the database (i.e., a maximum of 8 boolean connectors per query).

Figure 22 summarizes the selection process considering the database search. Initially, 15,960 papers were assessed. Observe that this number differs from the total presented in Table 21. This situation occurred because Wiley limits the number of references to 2,000. Therefore, the authors could not access the remaining 488 studies that Wiley retrieves.

Table 21 – Number of studies retrieved per base.

Database	Studies per Base
IEEE Xplore	2,875
ACM Digital Library	1,204
Scopus	3,965
Web of Science	1,795
ScienceDirect	145
Engineering Village	3,976
Wiley	2,488
Total	**16,448**

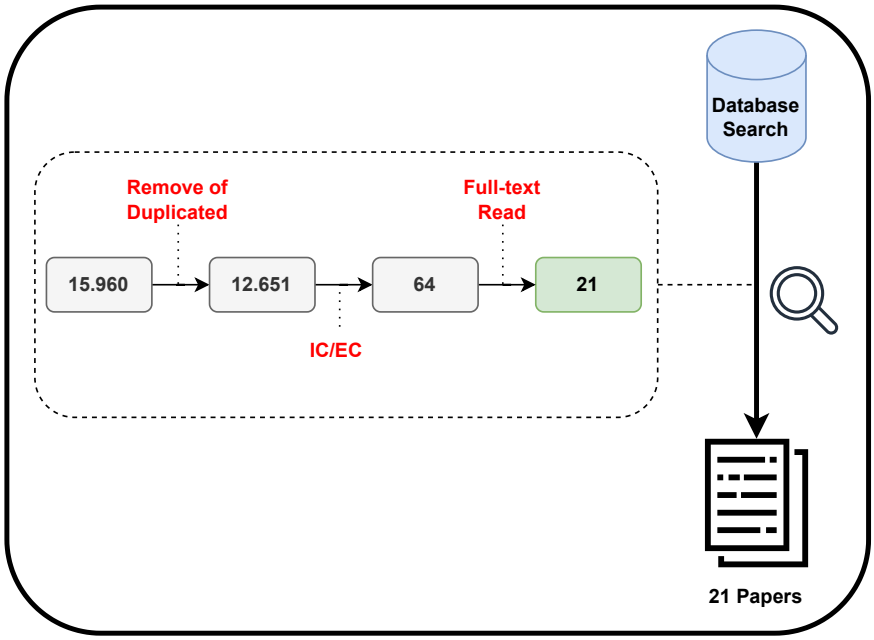


Figure 22 – Study selection process regarding the database search.

After removing the duplicates, we had 12,651 studies for assessment. Next, the IC/EC was applied, and 64 papers were identified as inclusion candidates. Finally, the authors read the 64 papers and defined a preliminary set of 21 papers for the snowballing phase.

Figure 23 presents detailed information about the snowballing phase. In the first round, 760 papers were assessed (210 from forward snowballing and 550 from backward snowballing). After evaluation, four new papers were found for inclusion (2 from forward snowballing and two from backward snowballing). Next, the second round of snowballing was performed considering these four papers. The authors assessed 306 papers and no new studies were found for inclusion.

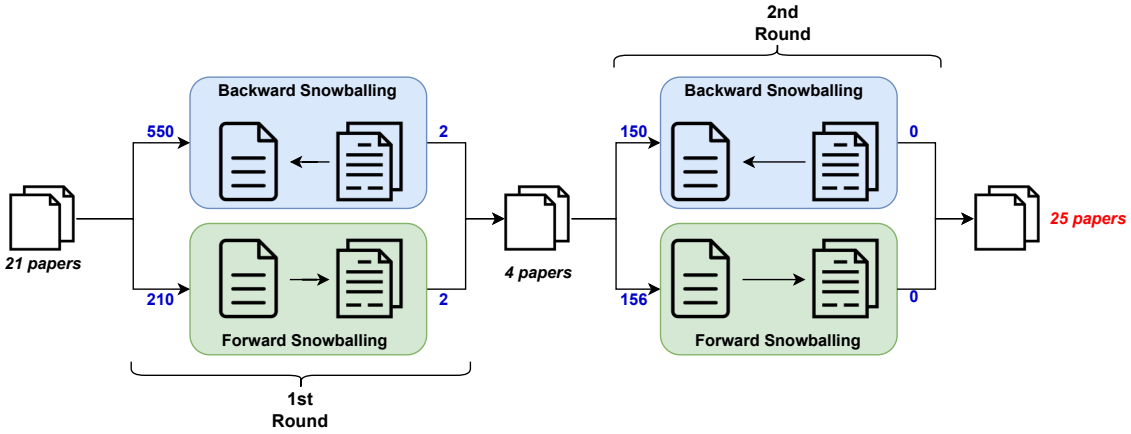


Figure 23 – Snowballing phase summarization.

At the end of the process, 25 papers were included in the mapping study. As described before, in **February 2024**, the authors updated the search in the digital libraries by reapplying the search string. In this case, the studies were filtered from 2023 until February 2024.

This search brought two new studies that were added to the final set. Moreover, backward and forward snowballing were applied and no new candidate papers were found. Hence, the final set of included studies comprises 27 primary studies as presented in Table 22. Moreover, the authors provide a repository with all relevant information to conduct the systematic mapping study[2].

[2] <https://bit.ly/mobile-application-testing-infrastructure>

Table 22 – List of all included studies in the systematic mapping study.

ID	Reference	Title
S1	(LIU, 2019)	A Compatibility Testing Platform for Android Multimedia Applications
S2	(ZHANG et al., 2017)	Towards A Contextual and Scalable Automated-Testing Service for Mobile Apps
S3	(HUANG, 2014)	AppACTS: Mobile App Automated Compatibility Testing Service
S4	(ZHANG; PI, 2015)	Mobile Functional Test on TaaS Environment
S5	(LIANG et al., 2014)	Caiipa: Automated Large-Scale Mobile App Testing through Contextual Fuzzing
S6	(MILTENBERGER et al., 2020)	DFarm: Massive-Scaling Dynamic Android App Analysis on Real Hardware
S7	(LUO et al., 2019)	CamTest: A Laboratory Testbed for Camera-Based Mobile Sensing Applications
S8	(FARIA et al., 2019)	On Using Collaborative Economy for Test Cost Reduction in High Fragmented Environments
S9	(DHANAPAL et al., 2012)	An Innovative System for Remote and Automated Testing of Mobile Phone Applications
S10	(TAO; GAO, 2017)	On building a cloud-based mobile testing infrastructure service system
S11	(TAO; LIN; LU, 2015)	Cloud platform based automated security testing system for mobile internet
S12	(XAVIER et al., 2017)	Mobile Application Testing on Clouds: Challenges, Opportunities and Architectural Elements
S13	(GUO et al., 2018)	FeT: Hybrid Cloud-Based Mobile Bank Application Testing
S14	(BINH et al., 2020)	Experience Report on Developing a Crowdsourcing Test Platform for Mobile Applications
S15	(WU et al., 2017)	AppCheck: A Crowdsourced Testing Service for Android Applications
S16	(MA et al., 2016)	An Automated Testing Platform for Mobile Applications
S17	(VAJAK et al., 2018)	Environment for Automated Functional Testing of Mobile Applications
S18	(ROJAS; MEIRELES; DIAS-NETO, 2016)	Cloud-Based Mobile App Testing Framework: Architecture, Implementation and Execution
S19	(LANUI; CHIEW, 2019)	A Cloud-Based Solution for Testing Applications' Compatibility and Portability on Fragmented Android Platform
S20	(ALI; MAGHAWRY; BADR, 2018)	Automated parallel GUI testing as a service for mobile applications
S21	(AMANO et al., 2018)	Smartphone Applications Testbed Using Virtual Reality
S22	(PRATHIBHAN et al., 2014)	An automated testing framework for testing android mobile applications in the cloud.
S23	(KAASILA et al., 2012)	Testdroid: automated remote UI testing on android
S24	(SUN et al., 2023)	Taming Android Fragmentation Through Lightweight Crowdsourced Testing
S25	(STAROV; VILKOMIR, 2013)	Integrated TaaS platform for mobile development: Architecture solutions
S26	(KUROISHI; MALDONADO; VINCENZI, 2023)	Towards the Implementation of a Mobile Application Testing Infrastructure at Von Braun Labs
S27	(LIN et al., 2023)	Virtual Device Farms for Mobile App Testing at Scale: A Pursuit for Fidelity, Efficiency, and Accessibility

3.5 Results

3.5.1 Answers to RQ1

3.5.1.1 Year of publication, the authors and venues of primary studies

Figure 24 shows the distribution of publications per year. As can be observed, the first two primary studies that propose mobile application testing infrastructure were published in 2012 (S9 and S23). The years 2017, 2018, and 2019 had the highest number of publications, four papers per year. On the other hand, in 2021 and 2022, we could not find any publications related to mobile application testing infrastructure. Observe that 81% of the studies were published from 2012 to 2019. Moreover, 70% of the studies were published between 2014 and 2019. Considering the last four years – from 2020 to 2023 – five papers were published (see Figure 25). In 2020, two studies were found, and in 2023, three were found.

Next, Table 23 presents the number of studies published per venue (i.e., conference, journal, workshop, and symposium). As can be seen, sixteen papers (59%) papers were

Table 23 – Frequency of studies per type of venue.

Type of Venue	Number of Studies	Study ID
Conference	16	S1, S7, S8, S10, S11, S20, S24, S27
Journal	7	S3, S5 S6, S9, S12, S13, S14, S15, S16, S17, S19, S21, S22, S23, S25
Symposium	3	S4, S18, S26
Workshop	1	S2

published in conferences. Seven papers (25%) were published in different journals. Three papers were published in symposiums (11%), and one paper was published in a workshop (3%). Observe from Table 24 that some papers were published in renowned and well-

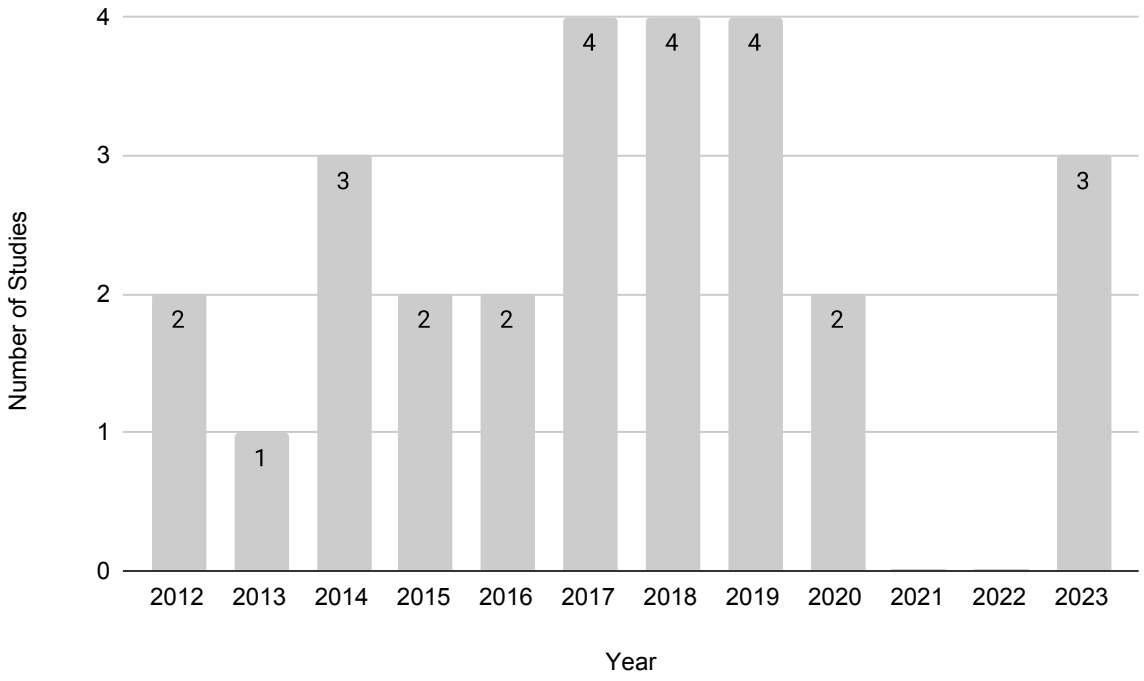


Figure 24 – Number of studies published per year.

known venues such as IEEE Transactions on Software Engineering, Journal of Systems and Software, Future Generation Computer Systems, Annual International Conference on Mobile Computing and Networking, and Multimedia Tools and Applications.

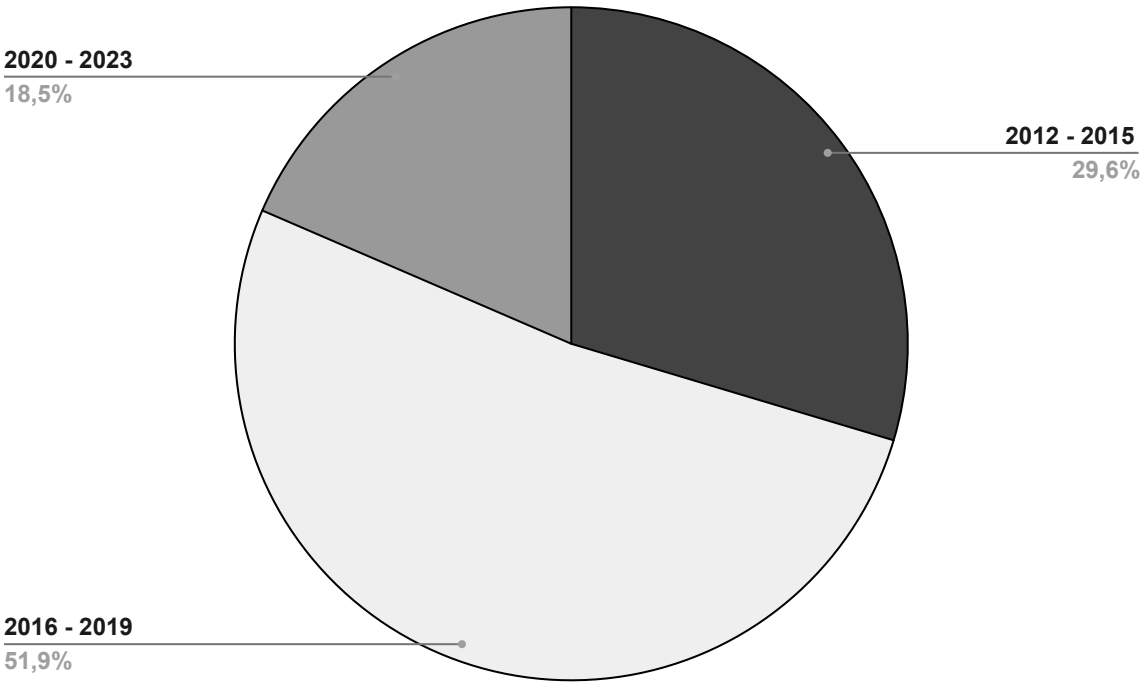


Figure 25 – Number of studies published per quadrennium.

Table 24 – Venues of the 25 included studies.

ID	Type	Venue
S1	Journal	Multimedia Tools Application
S2	Workshop	International Workshop on Mobile Computing Systems and Applications
S3	Conference	IEEE International Conference on Mobile Cloud Computing, Services, and Engineering
S4	Symposium	IEEE Symposium on Service-Oriented System Engineering
S5	Conference	Annual International Conference on Mobile Computing and Networking
S6	Conference	International Conference on Mobile Software Engineering and Systems
S7	Journal	Pervasive and Mobile Computing
S8	Journal	Future Generation Computer Systems
S9	Conference	Service Research and Innovation Institute Global Conference
S10	Journal	The Journal of Systems and Software
S11	Journal	Tsinghua Science and Technology
S12	Conference	Euromicro International Conference on Parallel, Distributed and Network-Based Processing
S13	Conference	IEEE International Conference on Software Quality, Reliability and Security Companion
S14	Conference	International Conference on Computational Collective Intelligence
S15	Conference	International Conference on Web Services
S16	Conference	IEEE International Conference on Software Quality, Reliability and Security Companion
S17	Conference	International Conference on Smart Systems and Technologies
S18	Symposium	Brazilian Symposium on Systematic and Automated Software Testing
S19	Conference	Asia-Pacific Software Engineering Conference
S20	Journal	Journal of Software: Evolution and Process
S21	Conference	EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services
S22	Conference	IEEE International Conference on Advanced Communication Control and Computing Technologies
S23	Conference	International Conference on Mobile and Ubiquitous Multimedia
S24	Journal	IEEE Transactions on Software Engineering
S25	Conference	International Conference on Automation of Software Test
S26	Conference	International Symposium on Software Reliability Engineering
S27	Conference	Annual International Conference on Mobile Computing and Networking

The 27 primary studies in the SMS encompassed 116 authors. Only three of the authors appeared in two different papers ((ZHANG et al., 2017), (LIANG et al., 2014), (LUO et al., 2019), (KAASILA et al., 2012), (FARIA et al., 2019), (KUROISHI; MALDONADO; VINCENZI, 2023)), showing a large diversity of authors that published studies focusing on proposing testing infrastructure for mobile application testing.

3.5.1.2 Quality of primary studies

This section presents the quality of the primary studies included in the SMS. For this study, two quality protocols were adopted (see Section 3.4.5). In this case, the goal is to present the quality of primary studies from an academic and an industry point of view. To evaluate the studies from an academic perspective, it was considered the protocol defined by Dybå e Dingsøyr (2008). Moreover, the protocol defined by Ivarsson e Gorschek (2011) was adopted to evaluate the quality of the studies from an industry perspective. Hereafter, we refer to the academic quality score as QAA and the industry quality score as QAI.

Table 25 presents the QAA findings. After analyzing the results, only studies S24 and S27 achieved the highest score of 11. Two studies (S8 and S15) have a quality score of 10; in four studies (S1, S5, S7, and S20), the score was 9. On the other hand, two studies had the lowest score of 2 (S14 and S22).

Table 25 – Quality assessment results considering an academic perspective.

Quality Assessment – Academia												
ID	**QA1**	**QA2**	**QA3**	**QA4**	**QA5**	**QA6**	**QA7**	**QA8**	**QA9**	**QA10**	**QA11**	**Total**
S1	•	•	•	•	•		•	•		•	•	9
S2	•	•	•	•			•			•	•	7
S3	•	•								•	•	4
S4	•	•				•				•	•	5
S5	•	•	•	•		•	•	•		•	•	9
S6	•	•					•	•		•	•	6
S7	•	•	•	•		•	•	•		•	•	9
S8	•	•	•	•	•		•	•	•	•	•	10
S9	•	•	•	•			•				•	6
S10	•	•	•	•			•	•		•	•	8
S11	•	•	•	•			•				•	6
S12	•	•								•	•	4
S13	•	•	•	•						•	•	6
S14		•	•									2
S15	•	•	•	•	•	•	•	•		•	•	10
S16	•	•	•								•	4
S17	•	•	•	•							•	5
S18	•	•	•	•							•	5
S19	•	•	•		•		•			•	•	7
S20	•	•	•	•		•	•	•		•	•	9
S21	•	•	•	•			•	•		•	•	8
S22	•			•								2
S23	•	•	•								•	4
S24	•	•	•	•	•	•	•	•	•	•	•	11
S25	•	•	•	•							•	5
S26	•	•	•	•	•		•			•	•	8
S27	•	•	•	•	•	•	•	•	•	•	•	11

Most studies fulfill the quality criteria QA1, QA2, QA3, QA7, QA10, and QA11. On the other hand, few studies precisely describe the subject cases (QA5) and present the threats to validity (QA9). Considering QA9, only S8, S24, and S27 explicitly describe and discuss the threats to the validity.

Next, the quality assessment results from an industry perspective were presented. The results are displayed in Table 26 and Table 27.

First, Table 26 presents the *rigor* score of each study, which considers three dimensions: Context, Design, and Validity threats. Recapping from Section 3.4.5, each attribute may have three possible values (0, 0.5, or 1), and the maximum score is three.

Observing the results, three studies (S8, S24, and S27) obtained the highest score of 3. Moreover, four studies (S5, S7, S10, and S15) obtained a score of 2.5. In this case, the studies do not present an in-depth discussion of the threats to validity. Study S22 has the lowest score of 0.5 because it briefly discussed the study context and lacked a description of the study design and the threats to the validity.

Note that two quality attributes are the opposite in terms of the appliance. On one hand, most studies properly describe the context in which the study was carried out. On the other hand, few studies have presented the threats to validity. Only S8, S24 and S27

Table 26 – Quality assessment results considering an industrial perspective – Rigor.

ID	Context			Design			Validity threats			Total
	S (1)	*M (0.5)*	*W (0)*	*S (1)*	*M (0.5)*	*W (0)*	*S (1)*	*M (0.5)*	*W (0)*	
S1	•			•					•	2
S2	•			•					•	2
S3		•				•		•		1
S4		•			•				•	1
S5	•			•				•		2.5
S6		•		•					•	1.5
S7	•			•				•		2.5
S8	•			•			•			3
S9	•				•				•	1.5
S10	•			•				•		2.5
S11	•				•				•	1.5
S12		•			•				•	1
S13	•				•				•	1.5
S14	•					•			•	1
S15	•			•				•		2.5
S16	•				•				•	1.5
S17	•				•				•	1.5
S18	•				•				•	1.5
S19	•			•					•	2
S20	•				•				•	1.5
S21	•				•				•	1.5
S22		•				•			•	0.5
S23	•				•			•		2
S24	•			•			•			3
S25	•					•				1
S26	•			•					•	2
S27	•			•			•			3

explicitly describe the possible threats to validity, while S3, S5, S7, S10, S15, and S23 briefly present them. Considering *Design* attribute, the studies S3, S14, S22, and S25 have no description or discussion about the study design.

The next analysis relies on the *Relevance* of each included study based on the *User/Subject*, *Context*, *Scale*, and *Research*. One key difference between *Rigor* and *Relevance* is that the value of the latter attribute is binary, i.e., the score assigned is one whether the attributes contribute to relevance and 0 otherwise. Table 27 presents the results.

According to the results, study S27 was the only one to score 4. The studies S1, S5, S7, S13, and S21 have score of 3 when considering *Relevance*. Contrarily, in 7 studies (S3, S4, S12, S18, S19, S22, and S25), the score obtained was 0, showing that these studies do not have industry relevance following the protocol of Ivarsson e Gorschek (2011).

The final quality score of the studies from an industry perspective consists of the sum of the rigor and relevance scores. Figure 26 summarizes the results.

Table 27 – Quality assessment results considering an industrial perspective – Relevance.

ID	User/Subject		Context		Scale		Research Method		Total
	CR (1)	*NCR (0)*	*CR (1)*	*NCR (0)*	*CR (1)*	*NCR (0)*	*CR (1)*	*NCR (0)*	
S1	•			•	•		•		3
S2		•	•			•		•	1
S3		•		•		•		•	0
S4		•		•		•		•	0
S5	•		•		•			•	3
S6	•			•	•			•	2
S7	•		•			•	•		3
S8	•			•		•		•	1
S9		•		•	•			•	1
S10	•			•		•	•		2
S11	•			•	•			•	2
S12		•		•		•		•	0
S13	•		•		•			•	3
S14	•			•		•		•	1
S15	•			•	•			•	2
S16		•	•			•	•		2
S17	•			•	•			•	2
S18		•		•		•		•	0
S19		•		•		•		•	0
S20	•			•		•		•	1
S21	•		•		•			•	3
S22		•		•		•		•	0
S23	•		•		•		•		4
S24	•			•	•		•		3
S25		•		•		•		•	0
S26		•	•		•		•		2
S27	•		•		•		•		4

Study S27 achieved the highest score of 7, followed by studies S23 and S24, which obtained a score of 6: S23 obtained a score of 2 considering *Rigor* and four considering *Relevance*, and S24 obtained a score of 3 considering *Rigor* and three considering *Relevance*. Two studies (S5 and S7) achieved a score of 5.5; in one study (S1), the score was 5.

The studies S3, S4, S12, S18, S22, and S25 have the lowest scores of 1, 1, 1, 1.5, 0.5, and 1, respectively. It is worth noting these studies did not score in terms of *Relevance*.

Next, we compared the quality studies considering both strategies. First, the data was normalized using the same base for a fair comparison. Therefore, a script was developed using scikit-learn Python library to normalize the data in an interval of [0,1], considering the final score obtained by the two strategies. Figure 27 presents the results of the data normalization, and for better data visualization, each score was rounded into two decimal places.

When assessing the results, the average quality score of the studies considering QAA was slightly better than the industry protocol. The average quality score of QAA was 0.52, whereas the quality score of QAI was 0.45, with a standard deviation of 0.28 in both cases.

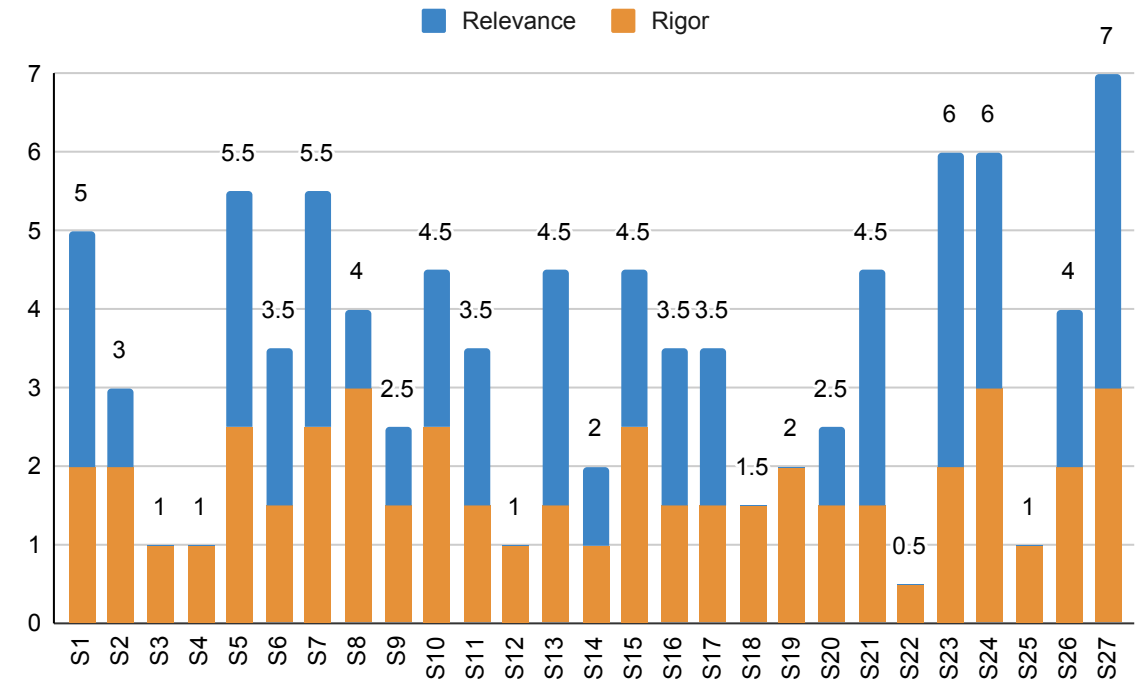


Figure 26 – Final quality score from the 25 studies considering an industrial perspective.

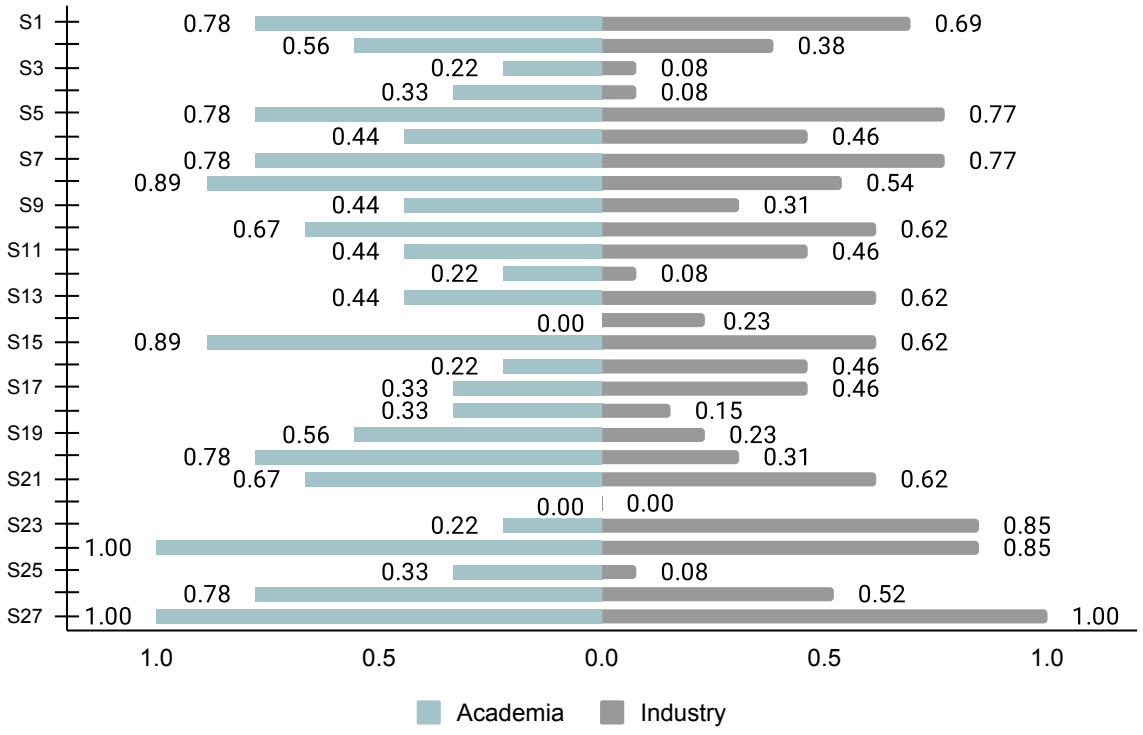


Figure 27 – Normalized quality score considering both strategies.

Next, the statistical difference between the samples was evaluated. The first step consisted of checking whether QAA and QAI had a normal distribution by applying the Shapiro-Wilk normality test since the sample size was less than 50 (RAZALI; YAP, 2011). For this purpose, the following hypotheses were defined:

- ❏ Null hypothesis (H_0): The samples QAA and QAI do not have a normal distribution.
- ❏ Alternative hypothesis (H_1): The samples QAA and QAI have a normal distribution.

The results showed that QAA had a *p-value* of 0.22 and QAI had a *p-value* of 0.31. Therefore, there was insufficient statistical evidence to reject the null hypothesis; hence, the samples had a normal distribution.

Next, the Student's t-test computed the statistical differences between QAA and QAI. For this purpose, the following hypotheses were defined:

- ❏ Null hypothesis (H_0): There is no difference between QAA and QAI.
- ❏ Alternative hypothesis (H_1): There is a difference between QAA and QAI.

The final *p-value* obtained was 0.0000026064. the null hypothesis (H_0) was rejected since the *p-value* < 0.05. Therefore, there is a statistical difference (H_1) between QAA and QAI. The 27 included studies had a better quality score from an academic view rather than from an industry perspective.

An interesting observation is that the type of venue (i.e., journal or conference) does not necessarily impact the quality of the studies. When assessing the ten most high-quality studies considering QAI, we observed that six were published in journals, and four were published in well-known conferences. Similarly, the assessment of QAA showed that six studies were published in well-known conferences, while four were published in journals.

3.5.2 Answers to RQ2

3.5.2.1 Definition of a new classification for mobile application testing infrastructure

In Section 3.3 the concepts of testing infrastructure for mobile application testing defined by Gao et al. (2014) were presented. In this case, we partially adapted these concepts and envisioned a new classification. The *cloud* and *crowdsourcing* terminologies were adopted, but instead of using the *device-based* and *emulator-based*, we defined a new type named **Local**. Figure 28 depicts a mind map of the new classification.

For **cloud**, the same concept was adopted, i.e., a testing infrastructure that benefits from cloud resources to build a mobile device cloud. For **crowdsourcing**, the concept of Gao et al. (2014) was extended considering those testing infrastructures that leverage testing, engineers, users, and community to run the tests, and also considering those that rely on idle devices/available devices from real users to run the tests.

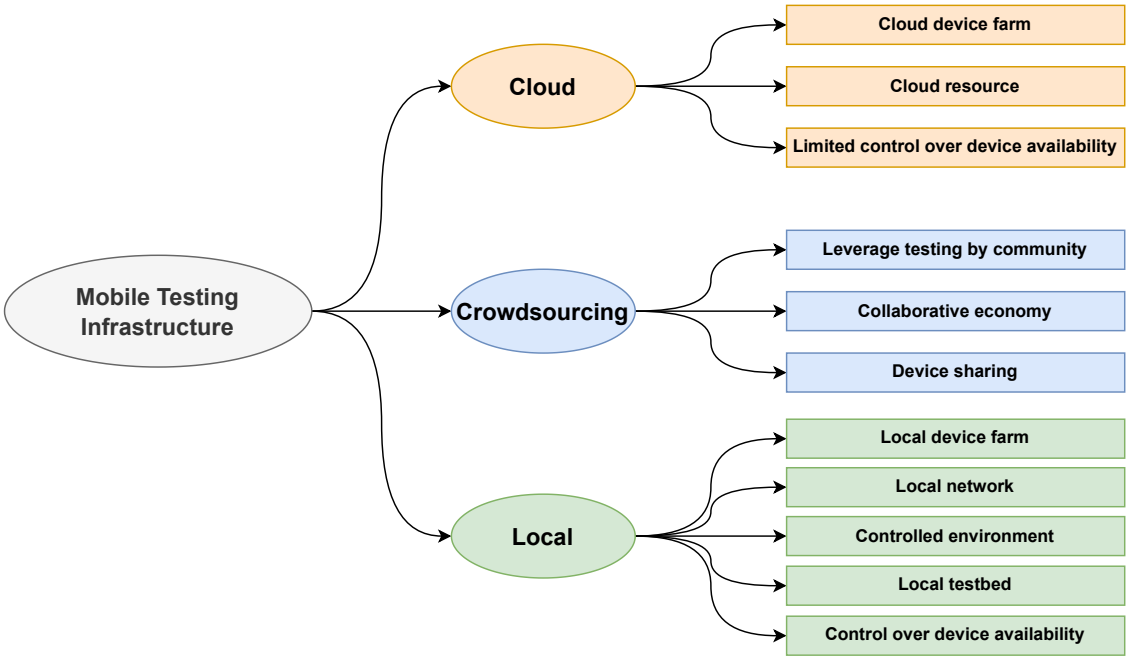


Figure 28 – Mind map of the new classification.

Local refers to the infrastructures designed to run in a local setting, such as a laboratory setting, a local testbed, a local infrastructure that runs specific types of testing, or a local device farm. Observe that the **local** infrastructure differs from the other two presented by Gao et al. (2014) (i.e., emulator and device-based) because it focuses on the characteristic of the infrastructure rather than the type of the devices.

3.5.2.2 Types of Testing Infrastructure

This section presents the results of 2a. To answer this question, the included studies were assessed based on the types of testing infrastructure proposed according to the classification presented in the previous section. Hence, the 27 studies were classified into three possible categories: Cloud, Crowdsourcing, and Local. Figure 29 summarizes the findings regarding types of testing infrastructure.

As can be seen, the majority of studies propose cloud-based testing infrastructure. In total, 17 studies (S1, S2, S3, S4, S5, S6, S10, S11, S12, S13, S16, S18, S19, S20, S22, S23, and S25) were categorized as *Cloud*, totaling 63% of the studies.

In S1, Liu (2019) proposed a cloud-testing platform for Android multimedia applications to be tested automatically against a large variety of physical devices, where the users upload a test script for the target app and select a set of Android devices. As a result of the tests, the infrastructure provides a report providing bug information, device hardware, network capabilities for the applications, visual (i.e., video and screenshots) information about the test execution, performance data, and app crash data. Moreover,

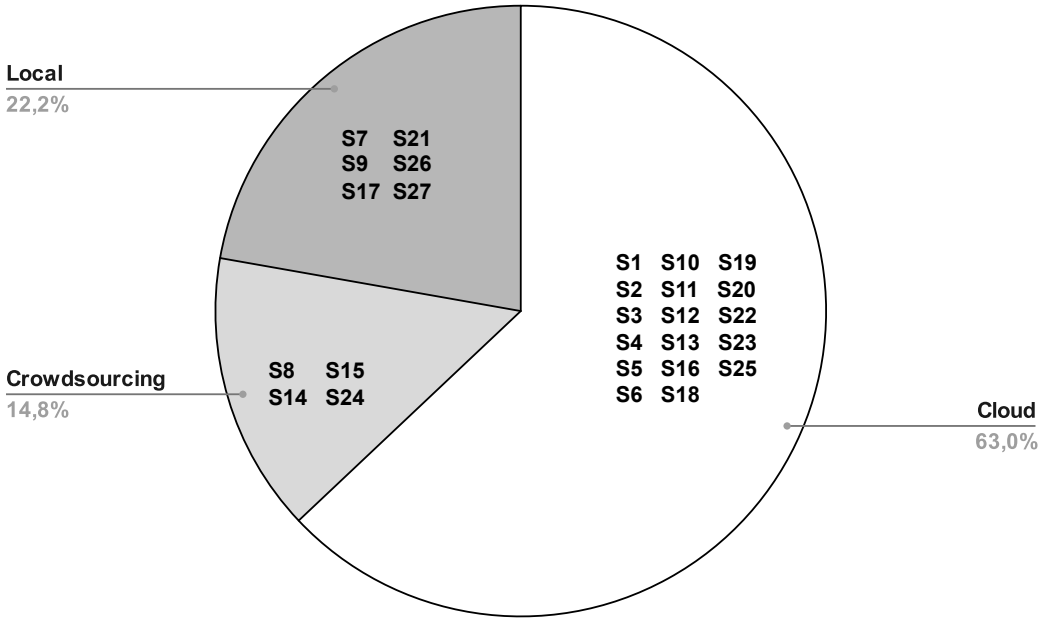


Figure 29 – Distribution of the studies per types of testing infrastructures.

the infrastructure is designed to run the tests using existing tools (i.e., Installer, Monkey, ACE, Monkeytalk, Robotium, Robot Framework, UiAutomator, and Espresso) rather than proposing a specific tool to test multimedia features.

In S2, Zhang et al. (2017) envision a narrow-waist architecture, namely RainDrops, that splits the app execution load between the cloud container and physical devices. That is, this architecture leverages both emulators to run tests at the UI level and physical devices to run tests that require context-sensitive system API calls. Furthermore, a prototype was developed to assess the feasibility of the approach. The app developers submit the Android app binary through a cloud portal, the test specification file containing information about the real-world contexts (geolocation, network conditions, etc.), and an optional custom test driver.

In S3, Huang (2014) proposed an automated compatibility testing service named AppACTS. A user uploads the application under test and selects the device for testing through a web user interface. The devices are organized as a mobile device pool that could be geographically distributed. Moreover, a module that sends action sequences (i.e., touch, press, drag, type) to test the application provides a fully automated testing process.

The study of Zhang e Pi (2015) (S4) proposed a TaaS (Testing-as-a-Service) platform for Android apps. In summary, the platform supports five main steps: (i) upload of a native app under test by the user; (ii) test script generation (or the user may provide a script with the actions to be performed); (iii) device selection for testing and test environ-

ment customization responsible for configuring the device according to location, language, network speed, etc; (iv) test device management which is responsible for executing the tests scripts in each mobile device; and (v) test results generation. One key contribution of the paper relates to the test script generation based on functional traversal, which relies on the weight of each command to provide a set of actions to be performed during the tests. Moreover, the TaaS platform has a module that collects context data from the user and rewards the user for the contribution.

In S5, Liang et al. (2014) proposed Caiipa, a prototype cloud service for testing context-aware apps in scale, using a contextual fuzzing approach. To test an application, the developers submit binaries of their apps. Then, Caiipa tests the application under various real-world contexts. It is worth noting that the test cases are ordered according to the probability of identifying crashes and performance issues.

In S6, Miltenberger et al. (2020) presented DFarm, a software and hardware system to configure and control Android devices in a private testing cloud, aiming at scalability and reducing the time required to conduct large-scale app-based analyses. Basically, the client interacts with the main controller, which is characterized as a desktop computer and is responsible for managing the devices. The main controller is connected to various sub-controllers through an internal Ethernet-based network. These sub-controllers may be desktop computers or single-board computers with USB hubs that connect mobile devices.

The study of Tao e Gao (2017) (S10) describes the development of a mobile infrastructure service system (or a mobile Infrastructure-as-a-Service). The paper provides a broad perspective of the need for a mobile cloud testing-as-a-service, presenting the benefits, issues, and limitations of current MTaaS and the key components of an MTaaS cloud service.

Tao, Lin e Lu (2015) (S11) presented a cloud-based platform focusing on security testing. Overall, the platform has three modules: web front-end module, testing environment construction module, and automated testing module. The first module provides a web interface in which the user defines the testing resource configuration and has access to the account management and testing results. The second module prepares the virtual hosts and the tools to be tested. It is worth noting that the platform uses the Metasploit testing framework to carry out the security testing. Finally, the third module is responsible for triggering the automated testing execution.

In S12, Xavier et al. (2017) proposed a cloud-based client-server system architecture for mobile application testing to make testing infrastructure more cost-effective. In the client module, the user submits the hardware and software requirements, the app under test, and the test script. The server module comprises three parts: client controller, system controller, and system monitor. The client controller provides all the information about

the test cycles. The system controller is responsible for managing the mobile emulators and cloud resources. Finally, the system monitor provides information about the testing progress and results.

Guo et al. (2018) (S13) presents FeT, a mobile hybrid-cloud service for testing Bank applications. The study presents an approach based on testing in zones that rely on the cloud property (i.e., public or private) and types of testing in different development phases. For instance, Zone 1 focuses on testing the behavior of the application using a public cloud. Zone 2 and Zone 3 are considered private clouds, but they differ in the types of testing applied in each phase. Using a public cloud reduces the need to acquire many mobile devices. The private cloud may provide a secure infrastructure that a bank application requires.

In S16, Ma et al. (2016) presented an automated testing platform named BugRocket, aiming at fragmentation issues. The platform implements a master-slave architecture. The master server provides a user interface in which the user submits the Android applications and publishes test requests. Moreover, the server is responsible for distributing the test request to the mobile devices connected to the slave servers. The slave server monitors and manipulates the mobile devices and sends the testing results to the master server.

Rojas, Meireles e Dias-Neto (2016) (S18) present AM-TaaS, a mobile cloud testing framework that facilitates the test environment setup and configuration aiming at testing in various mobile devices and platforms. The proposed framework has three main components: (i) a UI layer in which the user submits the app binary files and the test scripts and presents information about the test results; (ii) an information layer responsible for storing the files related to the test execution (i.e., the app binary file, the test scripts, information about the emulators, user data, etc); and (iii) a resource layer that is responsible for managing and starting the emulators and devices, execute the tests scripts, and to provide the test results for the user.

The study of Lanui e Chiew (2019) (S19) presents a cloud testing model to provide an automated application testing service. The model has two parts. The first part (Main Control Node) consists of the test management (i.e., list of available devices, devices command communicator, test script management) and the test report management. The second part (Devices Cluster Controller Node) manages the communication between the Main Control Node and the Android devices. In this layer, the devices are prepared, the tests are executed, and the results are collected and sent back to the Main Control Node. The proposed model is designed to be configured on a public, private, or hybrid cloud.

In S20, Ali, Maghawry e Badr (2018) proposed a cloud-testing service for mobile application GUI testing. The proposed system aims to perform all testing activities in multiple virtual nodes. The user uploads the application under test. Then, the system is responsible for test case generation and test execution. Concerning test case generation,

the authors relied on model-based testing techniques to generate a sequence of GUI actions that mimic the user behavior. For test execution, the approach relied on Appium to execute tests on multiple virtual nodes (or emulators).

Concerning S22, Prathibhan et al. (2014) presented a cloud-based mobile application testing framework for Android application testing in various mobile devices. The user uploads the application under test in APK format, and the infrastructure is responsible for running the tests on multiple devices. The authors proposed Mobile Application Testing Tool (or MAT Tool) that supports functional, performance, and compatibility testing.

The study of Kaasila et al. (2012) (S23) presented Testdroid, an online platform for scripted-based UI tests on various Android devices. Testdroid provides an online interface where the user uploads the APK under tests and the test script. This test script can be created using their test recorder tools that track the manual interaction of the user and the application. Then, the user selects the subset of devices to run the test scripts, and at the end of the process, the results are presented in the online interface.

In S25, Starov e Vilkomir (2013) proposed Cloud Testing of Mobile Systems (or CTOMS), a Testing-as-a-Service platform for testing the functionality of mobile applications on various connected devices. CTOMS can be divided into two parts. The first part (Node application) manages the devices, performs tests, and manages the interconnections with the second part (Main part). The latter is responsible for managing all the node applications and the end-user communications with the node applications.

Considering *Crowdsourcing* four studies (16%) were found: S8, S14, S15 and S24.

In S8, Faria et al. (2019) proposed a disruptive platform, Distributed Bug Buster (or DBB), to perform UI tests based on Espresso on multiple idle devices, using the concept of Collaborative Economy. Generally, the device owner makes their handset available for testing and, as a counterpart, receives a reward. The advantages of the approaches relate to the cost reduction of mobile application testing compared to existing testing services (for instance, AWS Device Farm and Google Firebase Test Lab) and the possibility of having a vast catalog of device models available for testing.

In S14, Binh et al. (2020) provides an experience report on developing a crowdsourcing platform (TMACSTest) for mobile application testing. The authors provide a brief overview of the platform development state and describe the platform's expected goals.

Wu et al. (2017) (S15) proposed AppCheck, a crowdsourced Android application record and replay testing service. Overall, AppCheck has a web interface in which real users interact with an application under test, and the crowdsourced event traces (or actions) are collected. Then, the event traces are executed across real users' devices that participate in crowdsourced testing. AppCheck has an oracle-checking algorithm that identifies performance and compatibility issues.

In S24, Sun et al. (2023) presented LazyCow, a lightweight crowdsourced testing approach focusing on taming Android fragmentation issues. This approach leverages idle devices of real users to execute the test cases and return the results to the server. Moreover, the authors proposed a lightweight approach that dispatches single test cases containing only a small piece of code snippets rather than the whole application.

Concerning *Local*, six studies (16%) were found: S7, S9, S17, S21, S26 and S27.

Luo et al. (2019) (S7) presented CamTest, a laboratory testbed for camera-based mobile sensing applications. The main idea was to propose a testing infrastructure to conduct realistic tests using replayed video images and video clip frames as test cases, providing a cost-effective solution when testing camera-based mobile sensing applications.

In S9, Dhanapal et al. (2012) presented a local testing system where the testers control the handset over the Internet. The system has a 3-axis robotic system that responds to the input provided by the user, i.e., the actions provided by the user are performed on the handset by the robotic system. The output of the tests is captured through a webcam, and the image frames are sent to the user.

In S17, Vajak et al. (2018) proposed a mobile application testing environment to test the UI interface of iOS and Android applications. This solution is designed to use the open-source Appium tool and has two modules: server and client. The server module is deployed in one machine and has Appium Desktop installed and configured. Appium Client is installed in the client module, and two Python scripts help communicate with the server. It is worth noting that the server executes the test and provides the test results for the user.

The study of Amano et al. (2018) (S21) presented an alternate testbed for smartphone applications that leverage a virtual reality architecture. The study aims to provide a virtual environment that simulates a physical world environment, and the application can be tested under real-world conditions.

In S26 Kuroishi, Maldonado e Vincenzi (2023), presents a practical experience report of implementing a testing infrastructure in an industrial scenario. The study first presented a comparison of existing commercial and open-source testing tools and services that are most suitable for implementation in the company. Moreover, the authors validate the testing infrastructure in a Continuous Integration/Continuous Deployment (CI/CD) environment.

Finally, in S27, Lin et al. (2023) presented a proposition and evaluation of a virtual device farm for large-scale testing of mobile apps. Like S26, the virtual device farm was implemented in a CI/CD environment and validated in an industrial scenario. The results show that a virtual device farm can be effective in mobile app testing at scale despite emulators having limited features compared to physical devices.

3.5.2.3 Types of Operating System

Figure 30 displays the distribution per operating system. Overall, three types of operating systems were identified: Android, iOS, and Windows Phone.

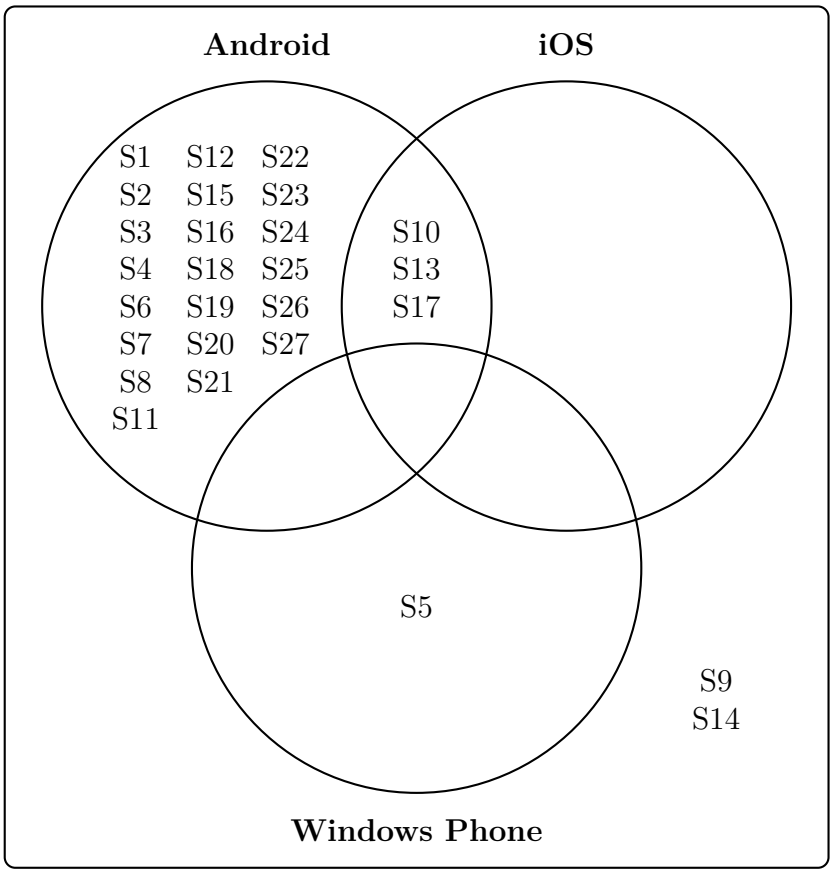


Figure 30 – Distribution of studies per mobile operating system.

According to the results, 19 studies (S1, S2, S3, S4, S6, S7, S8, S11, S12, S15, S16, S18, S19, S20, S21, S22, S23, S24, S25, S26 and S27) support the Android operating system in their testing infrastructure. Moreover, three studies (S10, S13, and S17) proposed infrastructure supporting Android and iOS. One study (S5) supports Windows Phone, and two studies (S9 and 14) do not explicitly address the supported operating system.

3.5.2.4 Types of Device Supported

Two types of devices can be used when testing a mobile application: *real devices* and *emulators*. Each of these two types of devices has pros and cons. For instance, emulators have advantages in terms of financial cost and scalability when compared to real devices (GAO et al., 2014). On the other hand, emulators have some limitations

that make testing some applications difficult. For instance, when considering an Android Bluetooth application, a native emulator may not be the most suitable choice since it does not offer support for Bluetooth[3].

According to Figure 31, 17 studies (S1, S3, S5, S6, S7, S8, S9, S13, S14, S15, S16, S17, S19, S21, S23, S24, and S25) proposed mobile testing infrastructure that supports real devices. Four studies (S11, S12, S20, and S27) proposed testing infrastructure that only supports emulators. In five studies (S2, 10, 18, S22, S26), the infrastructure supported both emulators and real devices. Finally, one study (S4) does not explicitly address the type of device the testing infrastructure supports.

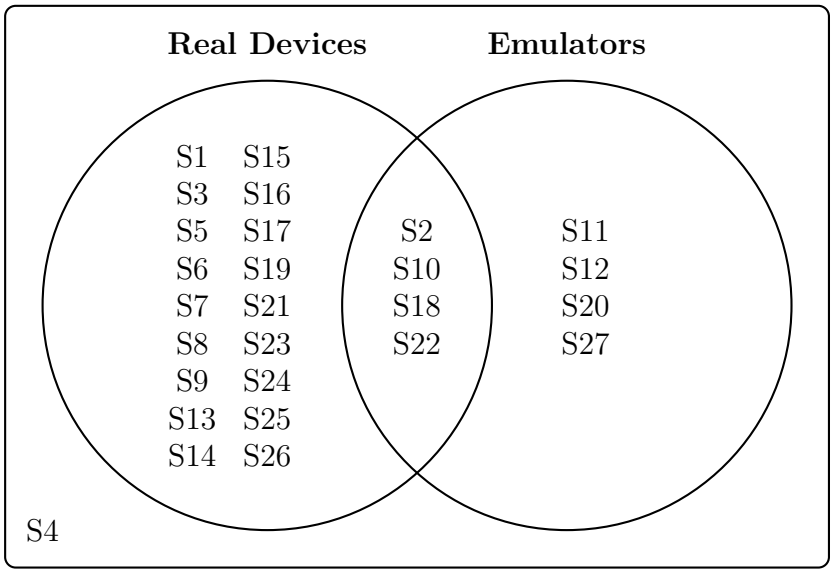


Figure 31 – Distribution of studies per types of devices.

3.5.2.5 Types of Testing Supported

This section presents the results considering the types of testing supported by each testing infrastructure. The results are summarized in Figure 32 and Table 28.

Figure 32 shows the frequency of types of testing addressed by the 27 included studies. *Functional testing* is the most considered type of testing appearing in 18 different studies, followed by *compatibility testing (10)*, and *performance testing (4)*. *Crash testing* and *install & uninstall testing* appeared in two different studies. Moreover, few studies support the other types of testing (acceptance, stress, usability, security, load, and smoke test).

Note that the number of types of testing addressed by the testing infrastructure (43, excluding *N/A*) surpasses the number of included studies (27). This happened because few testing infrastructures support more than one type of testing (see Table 28). S1 supports acceptance, stress crash, install & uninstall testing. S5 supports performance and crash testing. In S13, the supported types of tests are compatibility, performance,

[3] https://developer.android.com/studio/run/advanced-emulator-usage

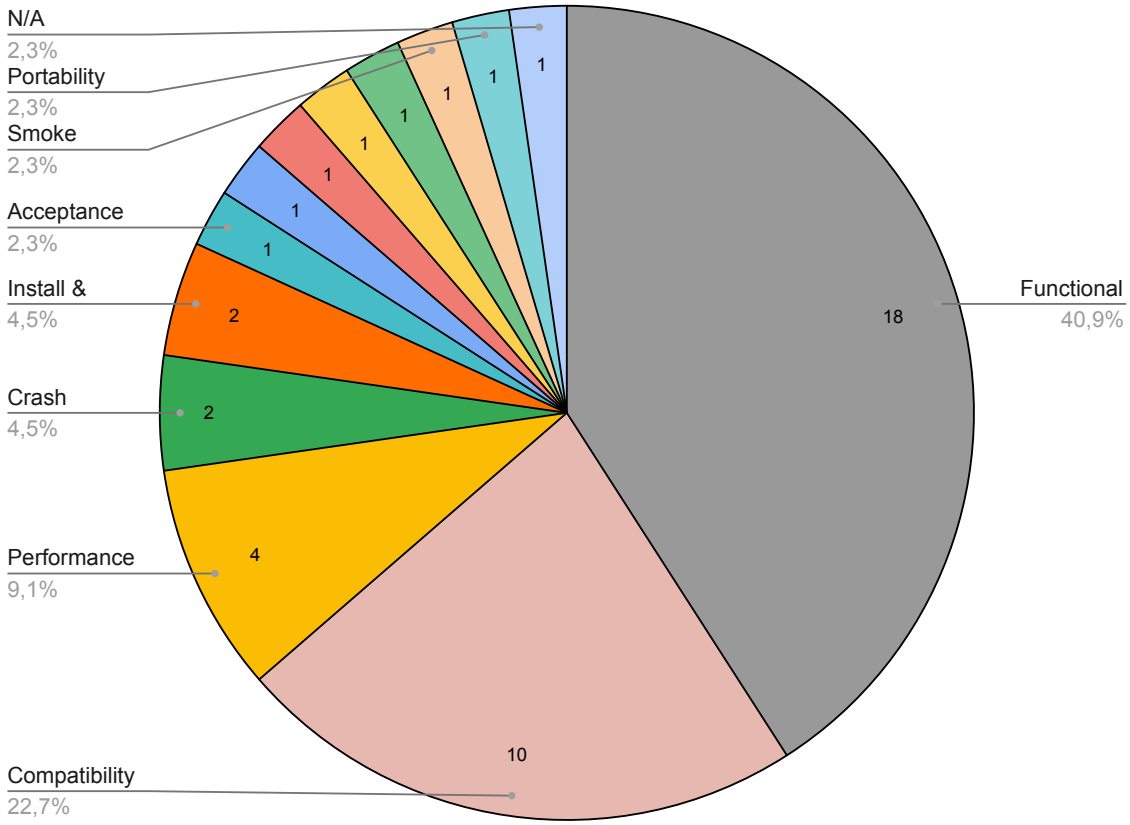


Figure 32 – Frequency of studies per type of testing.

load, functional, smoke, and install & uninstall testing. S16, S18, S26, and S27 support functional and compatibility testing. Finally, S22 supports functional, performance, and compatibility testing.

Table 28 – Distribution of studies per type of testing.

Type of testing	**Studies per types of testing**
Functional	S2, S4, S7, S8, S9, S10, S12, S13, S14, S16, S17, S18, S20, S22, S23, S25, S26, S27
Compatibility	S3, S13, S15, S16, S18, S19, S22, S24, S27
Performance	S5, S13, S15, S22
Crash	S1, S5
Install & Uninstall	S1, S13
Acceptance	S1
Stress	S1
Usability	S21
Security	S11
Load	S13
Smoke	S13
Portability	S19
N/A	S6

In twelve studies, the proposed testing infrastructure supports a single type of testing. Most of these studies focus on functional testing (S2, S4, S7, S8, S9, S10, S12, S14,

S17, S20, S23, and S25). In two studies (S3 and S24), the infrastructure is specific for compatibility testing. Finally, S11 focuses on security testing, and S21 focuses on usability testing.

3.5.2.6 Types of Application

Considering the types of applications supported by the testing infrastructure, five types were found: *Multimedia* (S1), *Context-aware* (S2, S4, S5, S21, S26), *Camera-based sensing* (S7), *Bank* (S13), and *Generic* (S3, S4, S6, S8, S9, S10, S11, S12, S15, S16, S17, S18, S19, S20, S22, S23, S24, S25, S26, S27). Note the *Generic* relates to those applications that are not from a specific domain or the study does not explicitly define the type of the application. Figure 33 presents the distribution type of application per study.

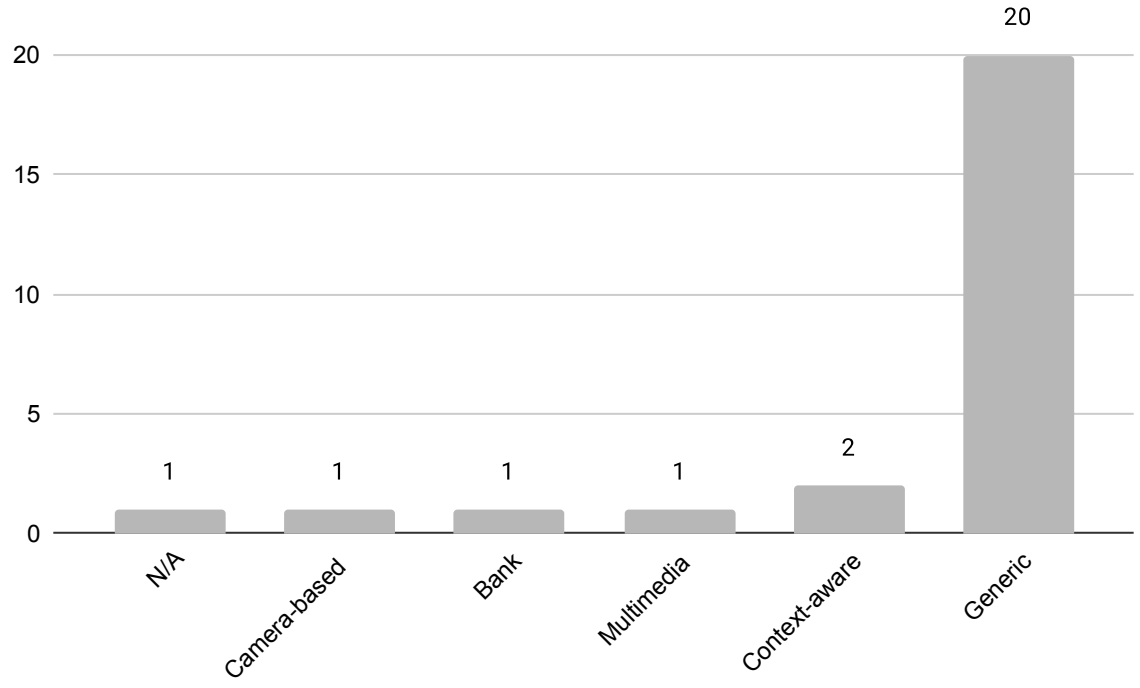


Figure 33 – Frequency of studies per types of applications.

Note that the number (26) presented in Figure 33 differs from the number of included primary studies. This happened because two studies (S4 and S26) propose a testing infrastructure that supports both *Context-aware* and *Generic* applications. In addition, S1 focuses on *Multimedia*, S7 presents a testbed for *Camera-based sensing* applications, and S13 study focuses on *Bank*. Moreover, in four studies (S2, S4, S5, and S21) the testing infrastructure address *Context-aware* applications. Finally, S14 does not explicitly address a specific application, and we could not identify the type of application supported by the infrastructure.

3.5.2.7 Availability for Usage/Experimentation

This section assesses whether the studies made their testing infrastructure available for usage or experimentation. This is important because (i) the artifact generated by the study can be applied or validated in an industrial scenario and (ii) for study replication.

Figure 34 summarizes the results. Observe that 23 studies (S1, S2, S3, S4, S5, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23*, S25, S26) do not provide or make available the testing infrastructure for usage/experimentation. Additionally, three studies (S6, S24, and S27) state that their solutions are available for download. It is worth noting that in the study S23, Kaasila et al. (2012) describes that the solution is available online for demonstration. However, the paper does not provide any link/URL to a repository or the demonstration, and after a quick search, we could not find any available online solution. Therefore, the study S23 was marked as "No" regarding availability.

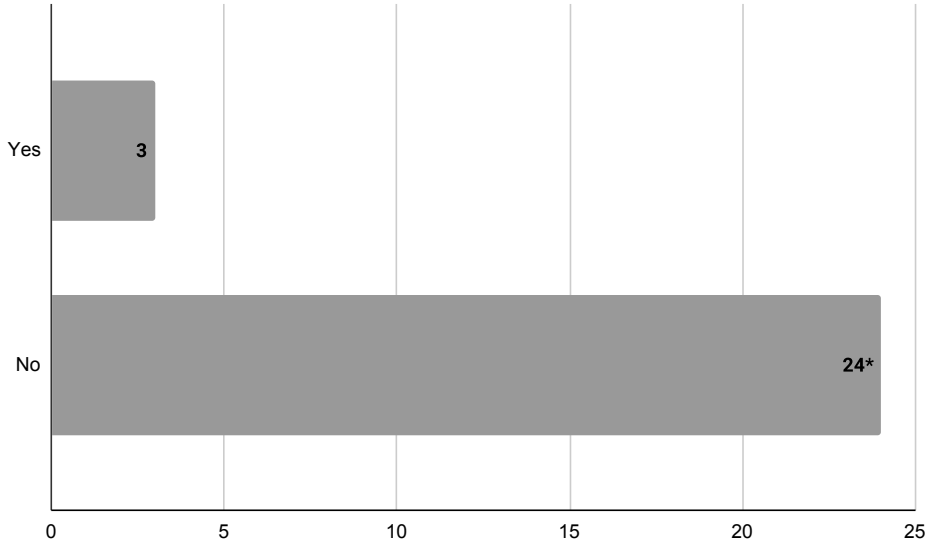


Figure 34 – Frequency of studies per availability.

3.6 Discussion and Future Directions

The previous section presented the results of the mapping study regarding existing testing infrastructure to support mobile application testing. Now, we provide a broad discussion presenting the main findings and the future directions for further research.

3.6.1 Discussion of the Results

The main goal of the paper is to provide a clear overview of the mobile testing infrastructure proposed in the scientific literature. To classify and categorize the existing types

of testing infrastructure, we partially relied on the taxonomy defined by Gao et al. (2014). In this case, the *Cloud* and *Crowdsourcing* terminologies were adopted as two possible infrastructures but also envisioned a third type named *Local*, which refers to those that infrastructures are designed to run on a local setting or laboratory.

According to the results, most studies (17) focus on the *cloud* by proposing different Testing-as-a-Service infrastructures. From a certain point of view, the results are already expected since TaaS may provide a scalable testing infrastructure with a reduced cost for the users (GIRARDON et al., 2020). In addition, *crowdsourcing* infrastructures started to gain more attention from the community over the last years. Therefore, it may be an interesting topic for further research since crowdsourcing testing infrastructure may also provide scalability. It can sometimes be cost-effective compared to existing cloud services (FARIA et al., 2019).

Considering *local* infrastructures, we found six studies that proposed this type of infrastructure. Local infrastructure may have some drawbacks compared to cloud and crowdsourcing, especially cost and scalability. However, depending on the type of application to be tested (for instance, a mobile application that may interact with external devices such as IoT devices or smart gadgets), creating a local infrastructure may be the most suitable option (KUROISHI; MALDONADO; VINCENZI, 2023). One advantage is the possibility of customizing the testing infrastructure according to the needs of the user/company.

Next, the types of operating systems supported by the testing infrastructure were assessed. As expected, the most supported OS is Android. In three studies (S10, S13, and S17), the infrastructure also supports iOS, and in one study, S5, the supported OS is Windows Phone. In general, Android OS is the most adopted by the academic community due to its open-source nature (TRAMONTANA et al., 2019). Despite the S5 focus on Windows Phone, the operating system has been discontinued since 2019[4].

The results show that most studies focus on real devices rather than emulators regarding the type of supported devices. Moreover, five studies (S2, S10, S18, S22, and S26) propose a hybrid infrastructure that supports real devices and emulators. In general, these two types of devices have their pros and cons. For instance, real devices may provide more realistic tests when compared to emulators, but the tests tend to be more costly due to the need to acquire the devices. Hence, we advocate that a hybrid infrastructure may be more cost-effective because a tester may leverage real devices and emulators to conduct the tests. For instance, a tester may use emulators to validate simple UI tasks and real devices to test different real-world conditions, e.g., various connectivity features of the applications (e.g., Bluetooth, NFC, RFID).

Next, the availability of the proposed testing infrastructure for usage or experimentation was assessed. The results show that most papers do not provide any form to access

[4] https://microsoftcaregh.com/2023/03/01/microsoft-phone-and-it-discontunation-know-why/

their solution. In summary, only four studies made their solution available. The studies S6, S24, and S27 provide a link to access their repository. Study S23 stated that the solution was available online; however, we could not find any information about the proposed platform. Since the study is from 2012, it was not possible to guarantee that the solution has already been discontinued or evolved into a different or commercial solution. We know that some barriers can make it difficult to share the solution with the academic community (KUROISHI; MALDONADO; VINCENZI, 2023). For instance, a study may be conducted in an academia-industry partnership, and the solution is protected by intellectual property rights (IPR). Hence, we expect the authors will not make their solution available to the academic community. However, in cases where the restriction does not apply, we advocate the importance of researchers making their solutions available to the community aiming at study replication or even a validation/implementation considering an industrial scenario.

Next, the results show that the 27 studies addressed 12 different types of testing, providing a heterogeneous scenario in terms of types of testing. That is, the 27 studies contribute different solutions that support a wide variety of types of testing rather than a specific type. Despite functional testing being the most considered, many studies propose solutions with various purposes.

Through the 27 included studies, most proposed testing infrastructure supports generic applications. On the contrary, some studies propose testing infrastructure that supports some specific application type. In this case, we identified four different domains of applications: multimedia, context-aware, camera-based, and bank. In a certain way, multimedia, context-aware, and camera-based applications may require a different or specific testing infrastructure because they rely on external events and/or specific hardware features provided by devices. Hence, we advocate for future researchers to keep exploring the proposition of different testing infrastructures to support these applications, especially those that rely on context or mobile device sensors (e.g., GPS, RFID, NFC, and Bluetooth).

Finally, the results were discussed regarding the quality of the studies. In this study, two types of quality assessment were performed: (i) a quality assessment considering an academic perspective and (ii) a quality assessment considering an industrial perspective. According to the results, the average quality score considering an academic perspective was slightly better when compared to the industrial perspective.

However, it is worth noting that regardless of the protocol applied, most of the included studies still lack quality. The average quality score was 6.70 out of 11 when considering the academic quality protocol. Similarly, when adopting the quality protocol of Ivarsson e Gorschek (2011), the average quality score was 3.48 out of 7. Therefore, we pinpoint the need to conduct more rigorous research and experimental studies concerning mobile application testing in the future.

3.6.2 Future Directions

The 27 primary studies in the systematic mapping study provided insights into how the area has evolved through the years and what the research gaps could benefit future research. Hence, we elicit some topics that may require further investigation.

Testing infrastructure to support multiple phases of the testing process:

When assessing the 27 included studies, we found that most testing infrastructures support only test execution and test reports. Some of these studies propose test case generation, especially for GUI testing. That is, the user uploads the binary of the application and the infrastructure to generate some GUI events to test the application. In this case, we advocate continuing to evolve the test case generation for UI and some parts of the application unrelated to GUI. For instance, test case generation for applications that use specific features of Android API and/or mobile devices that do not require GUI interactions.

Moreover, we pinpoint the need to keep investigating testing infrastructures that also support some testing criteria to assess the quality of the application or the test cases (e.g., coverage criteria or mutation testing).

Testing infrastructure to support different context-aware mobile applications:

Given mobile devices' ubiquitous and pervasive characteristics, context-aware mobile applications have gained considerable attention in the last few years. Hence, for future research, we suggest testing infrastructure that supports and optimizes context-aware mobile application testing, considering a laboratory setting and real-world scenarios.

Testing infrastructure to support different types of devices to optimize the test execution:

The results showed that most studies support real devices. We advocate a hybrid testing infrastructure that supports emulators and real devices, which may be cost-effective in optimizing test execution. By having a hybrid testing infrastructure, a tester/user can prioritize the parts of the application that are simple and do not require many resources to run on emulators and those that need a rigorous or specific type of test to run on devices.

The need to make the research artifact available to the academic community:

Making the testing infrastructure available to the community can bring many benefits to academia and industry. The academia may use the research artifact for replication

and validation. In addition, researchers may set up and use the testing infrastructure for experimentation. For instance, one can set up a testing infrastructure to validate the proposition of a new testing technique on multiple devices.

3.7 Threats to the Validity

This section presents and discusses the possible threats to the validity of the proposed systematic mapping study. Therefore, we highlight three main threats that may affect the results and how we tried to mitigate them.

The first threat to the validity relates to the *definition of the protocol*. In secondary studies, the definition of the protocol is fundamental because it encompasses all the relevant information to conduct the study, such as the study goal, the research questions, the study selection process, the inclusion and exclusion criteria, the quality assessment protocol, and the data extraction protocol. To mitigate the potential threats to the validity, the first author of the present paper defined the protocol, whereas the second, third, and fourth authors of the present paper reviewed each topic. In addition, we provided the study protocol in the repository for cross-checking [5].

The second threat to validity relates to the *study selection process*. To mitigate this threat, we performed a database search on well-known digital libraries and a manual search through backward and forward snowballing. In addition, we tried to generate a broad search string to retrieve the most relevant ones.

The first author of the present paper applied the inclusion and exclusion criteria during the study selection process. Whenever the metadata was insufficient, the authors discussed whether the paper should be included.

Finally, the third threat relates to the *results presentation*. Despite adopting a systematic process, the potential author bias may skew the final results. The first author of the present paper was responsible for extracting the relevant data. Next, the data collection was discussed among the other authors.

The quality assessment step may be subjective since it relies on the author's understanding. In this case, we adopted two protocols used in different secondary studies (YANG et al., 2021), and the findings were validated using a statistical test. QAA and QAI scores were normalized to the same base for a fair comparison. The Shapiro-Wilk normality test was used to check the normal distribution of the samples. The Student's t-test was also adopted to compute the statistical difference between QAA and QAI.

[5] <https://bit.ly/mobile-application-testing-infrastructure>

3.8 Conclusion

Although software testing is an essential activity for quality assurance and mobile application development, it is not yet widely adopted in the development phase (PECORELLI et al., 2021) and requires broader investigation from the academic and industrial community to make testing activities viable and cost-effective.

In this sense, this paper contributes to the area by providing a Systematic Mapping Study to categorize existing testing infrastructure to support mobile application testing. The first contribution is a new terminology to classify the testing infrastructures: **local**, **cloud**, and **crowdsourcing**. The results show that 63% propose cloud-based infrastructures, and recently, crowdsourcing-based has started to gain attention from the community.

The following finding showed that Android was the most considered operating system. In this case, this result was already expected due to the open-source nature of the OS. Regarding the type of devices, most of the testing infrastructure supports real devices, and only three infrastructures rely solely on emulators. Next, the results showed that few studies make their solution available for experimentation or usage, which is a drawback for the research area.

Moreover, the results showed that most studies support functional, compatibility, and performance testing. And considering the type of application, generic is the most supported by the testing infrastructure. In this case, a generic application is not from a specific domain (for instance, multimedia, context-aware, camera-based sensing, and bank), or the study does not explicitly define the type of the application.

We expect that the results of this study provide an overview for the research community on how the area of mobile application testing infrastructure is structured. To conclude, we highlight four research gaps for further investigation:

- ❏ The design of testing infrastructures to support multiple phases of the testing process.
- ❏ The design of testing infrastructures to support different context-aware mobile applications.
- ❏ The design of testing infrastructures to support different types of devices to optimize the test execution.
- ❏ The need to make the research artifact available to the academic community whenever possible.

Chapter 4

Towards the Implementation of a Mobile Application Testing Infrastructure at Von Braun Labs

4.1 Overview

The previous chapter presents a characterization of the state of the art regarding the existing testing infrastructure for mobile application testing. Despite the insightful results, the solutions offered in the literature would not meet the partner company's demands in the MAI/DAI project. The main reason concerns the characteristics of the company's application, as described in the paper. Thus, this paper aims to propose an alternative infrastructure that meets the company's requirements.

This chapter presents the definition of a testing infrastructure, or a local device farm, to support Android application testing. It presents a comparison of existing testing infrastructures and identifies the most suitable one for a real scenario. Below is the information of the paper (see also Figure 35):

- **Title:** Towards the Implementation of a Mobile Application Testing Infrastructure at Von Braun Labs.
- **Authors:** Pedro Henrique Kuroishi (UFSCar), José Carlos Maldonado (ICMC-USP), and Auri Marcelo Rizzo Vincenzi (UFSCar).
- **Local:** IEEE 34th International Symposium on Software Reliability Engineering (ISSRE).

❏ **Year:** 2023.

❏ **Status:** Published.

❏ **DOI:** <https://doi.org/10.1109/ISSRE59848.2023.00078>

2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)

Towards the Implementation of a Mobile Application Testing Infrastructure at Von Braun Labs

Pedro Henrique Kuroishi
Computing Department
Federal University of São Carlos
São Carlos, Brazil
phk@ufscar.br

José Carlos Maldonado
Institute of Mathematical and Computer Science
University of São Paulo
São Carlos, Brazil
jcmaldon@icmc.usp.br

Auri Marcelo Rizzo Vincenzi
Computing Department
Federal University of São Carlos
São Carlos, Brazil
auri@ufscar.br

Figure 35 – Information regarding paper P3.

4.2 Abstract

With the massive adoption of mobile devices, it became more mandatory for developers to provide high-quality applications. Nowadays, mobile devices are used for different purposes: entertainment, shopping, banking, and communication. Moreover, mobile devices can communicate and exchange information with various IoT devices distributed across the city. However, mobile application testing has different challenges when compared to other types of applications (i.e., desktop and client-server applications). First, we must consider mobile devices' different characteristics and limitations, such as connectivity, screen size, density, sensors, and limited battery. Second, there is a wide range of mobile devices from diverse vendors and models. Hence, there is a need to consider different device configurations to reduce compatibility issues that may occur in a high-fragmented ecosystem. In this case, several tools and services with various features and business models aim to run tests on multiple devices. In this practical experience report, we present the initial results of implementing a testing tool/service at Von Braun Labs to support the execution of tests across multiple Android devices. The stakeholders stated the need to (i) execute the tests on physical devices; and (ii) the tool/service must support tests that interact with a specialized IoT device. We start the study by comparing different tools/services to select the most suitable one for Von Braun Labs. We propose

a comparison framework to help evaluate six tools/services based on their technical, usability, and customization features. Then, we present a case study with an app from Von Braun Labs to validate the selected testing environment. Finally, we discuss the lessons learned, contributions, and future directions, pinpointing the need for a testing process since the beginning of the development project and the importance of lessening the gap between academia and industry.

4.3 Introduction

Over the last decade, mobile device users have grown significantly. One factor that explains this popularity is the possibility of performing multiple daily activities from a single smartphone. Mobile devices can be used for many purposes: communication, entertainment, shopping, banking, gaming, and tracking health status. Moreover, the concepts of *Industry 4.0* (GHOBAKHLOO, 2020) and *Smart Cities* (YIN et al., 2015) also transformed the way people interact with the environment making mobile devices crucial in this new paradigm (MATYI et al., 2020). Given this scenario, it becomes more mandatory for developers to deliver high-quality applications. Thus, software testing activities play a fundamental role in quality assurance.

Mobile application testing has several aspects that differ from other traditional applications (i.e., desktop and client-server applications) (MUCCINI; FRANCESCO; ESPOSITO, 2012). Mobile applications are designed to run on mobile devices. Hence, testing these applications must consider all the different characteristics and limitations of the devices, such as connectivity, screen size and density, sensors, and limited battery. In addition, some applications use context information to provide information or services to the user (i.e., mobile context-aware applications) (ALMEIDA; MACHADO; ANDRADE, 2020).

Another significant difficulty of mobile application testing relates to the fragmentation caused by various device models and operating versions. Considering the Android ecosystem, there are more than ten different OS version[1], and more than six different mobile device vendors[2]. Thus, testing an application with all possible combinations of OS and device models is impracticable (WEI; LIU; CHEUNG, 2016). In this case, there is a need to explore different forms of mobile application testing across multiple devices to reduce compatibility issues.

Using emulators is simpler to perform mobile application testing on multiple devices. However, some applications require using features not offered by the emulators, for instance, applications that use Bluetooth connection[3]. Using cloud services such as AWS

[1] https://www.statista.com/statistics/921152/mobile-android-version-share-worldwide/
[2] https://www.statista.com/statistics/271496/global-market-share-held-by-smartphone-vendors-since-4th-quarter-2009/
[3] https://developer.android.com/studio/run/advanced-emulator-usage

Device Farm (AMAZON, 2023), Firebase Test Lab (LLC, 2023), and Visual Studio App Center (CORPORATION, 2023) is a different way to perform distributed tests. These services have a pay-as-you-go business model in which they offer an infrastructure with various mobile devices, and the user pays for the minutes/tests used/executed in these services. The academic literature also presents some options to perform distributed tests across multiple devices, for instance, the tools of Malini et al. (2014) and Faria et al. (2019). Ultimately, the choice between a public cloud infrastructure and a local device farm may depend on the needs of the user/company. They need to understand the trade-offs related to the cost of each option (FARIA et al., 2019) and whether the features provided by these tools/services satisfy the scope and requirements of the application.

This practical experience report was carried out from an academia-industry collaboration. We describe the implementation of a testing environment (or local device farm) at Von Braun Labs to support mobile application testing across multiple Android devices. The applications developed by Von Braun Labs communicate with IoT devices located at different places. In this case, whenever the user approaches the IoT device and enters the tracking area, the mobile device (or smartphone) starts to communicate with the IoT device. Since the user is in movement, the communication between the smartphone and the IoT device must be instantaneous. Hence, the stakeholders stated that the testing environment has to support testing considering physical mobile devices due to the limitation of emulators and has to execute tests that interact with these specialized IoT devices. Given these requirements, we divided the study into two parts. First, we propose a comparative study to select the most appropriate tool/service for their environment. We defined a comparison framework to analyze the technical, usability, and customization features of each candidate tool/service. Next, we provide a case study to validate the deployed environment. Finally, we discuss the lessons learned, the contributions, and the future directions.

In summary, this paper makes the following contributions:

- ❏ A proposition of a comparison framework and a comparative study of six different tools/services to choose the most suitable for Von Braun Labs.
- ❏ A local device farm to execute tests that interact with specialized IoT devices.
- ❏ A case study to validate the proposed infrastructure.
- ❏ Proposition of a testing process in an ongoing project.

The paper is structured as follows: Section 4.4 describes all relevant information of the comparative study, whereas Section 4.5 displays and discusses the results. Section 4.6 presents the case study to validate the environment. Section 4.7 reviews the lessons learned and contributions. Section 4.8 discusses related works. Finally, Section 4.9 concludes the paper.

4.4 Comparative Study Setup

4.4.1 Problem Statement

The main goal of this comparative study is to help Von Braun Labs enhance their testing process by implementing a tool/service to facilitate the execution of tests across multiple devices. The tool/service will be applied to test applications that communicate and exchange information with specialized IoT devices. Due to intellectual property, we cannot present detailed information about the application.

Hence, the following research question is defined:

RQ: What is the most suitable testing tool/service for Von Braun Labs?

4.4.2 Comparative Study Description

To define the most suitable test tool/service for Von Braun Labs, we conducted a comparative study to collect all the useful information to help the decision process. The stakeholders define some requirements that the tool/service must have:

1. The tool/service has to execute tests in physical devices since emulators do not support many features.

2. The tool/service has to execute tests that interact with specialized IoT devices, which must be near the mobile devices and detected via sensors.

We started by defining a comparison framework to support the decision process (Section 4.4.3). This framework is similar to a checklist comprising the features of each tool/service. These features were derived according to the requirements defined by the stakeholders. The second step involves selecting a pool of tools/services as possible candidates (Section 4.4.4). Next, we choose five Android applications (Section 4.4.5) and two types of mobile devices (Section 4.4.6). Then, the applications were executed in each tool/service to extract all the relevant information defined in the comparison framework. Once all the data were collected, we presented it to the stakeholders and discussed the most useful tool/service.

4.4.3 Comparison Framework

Similar to the work of Amalfitano et al. (2022), which presents an evaluation and comparison of different Java mutation testing tools, the testing tools/service comparison framework was proposed based on their features. We analyzed and compared each tool/service according to the *Technical Features*, *Usability Features*, and *Customization*

Features. Each of these three main features comprises a set of specific characteristics that will be useful to support the decision process. These three main features were defined concerning the requirements defined by the stakeholders.

The Technical Feature category comprises the technical characteristics supported by each tool/service. The Usability Features group the specific features related to the usability of the tool/service, i.e., the characteristics related to the execution flow and ease of execution of the instrumented tests. Customization Features aim to assess the possibility of implementing future customization required by the stakeholders.

Table 29 presents detailed information, such as specific features, possible values, and a brief description of the comparison framework.

4.4.4 Tool/Service Selection

At the beginning of the project, the idea was to propose the use of Distributed Bug Buster (or DBB) (FARIA et al., 2019) in their testing environment for two main reasons: (i) it provides a disruptive way to execute the Android tests without the need to have a device connected to an Android Debug Bridge server, and (ii) it provides a low-cost alternative to execute test across multiple devices when compared to other testing services (the cost comparison can be found at (FARIA et al., 2019)). In this case, the stakeholders suggested a comparison of DBB and three existing cloud services, i.e., AWS Device Farm (AMAZON, 2023), Google Firebase Test Lab (LLC, 2023), and Visual Studio App Center (CORPORATION, 2023), to provide a clear overview of the advantages of using DBB.

However, there are some issues in DBB, primarily related to the current state of the tool. Many core dependencies need to be updated for broader usage. For instance, the main dependency to run DBB requires updating the support Library *Android Support* to *AndroidX*[4]. Another issue relates to the execution flow of DBB since many tasks require manual effort, i.e., giving proper permissions, accepting test plans, and app installation. Due to these limitations, we tried to find a more straightforward tool to execute the tests and provide an alternative for the stakeholders.

Spoon tool provides an easy way to perform tests across multiple devices. The main difference is that Spoon requires the device to be visible to the Android Debug Bridge (ADB) server[5] (i.e., the device should be connected to the server through USB or Wi-Fi). However, Spoon has the same problem as DBB since the tool is not currently maintained, and the dependencies also need to be updated before use.

After analyzing the issues tracker page of Spoon on GitHub, we found another alternative for the comparison study. The tool Marathon also supports the execution of tests across multiple devices. In addition, there are two advantages compared to DBB

[4] https://developer.android.com/jetpack/androidx
[5] https://developer.android.com/tools/adb

Table 29 – Description of the comparison framework

Type	Feature	Possible Value	Description
Technical Feature	OS Supported	Android / iOS	The mobile operating system supported by the tool/service.
	License	Open-source / Commercial / Prototype	Whether the tool/service is open-source, commercial, or a prototype.
	Last Release	Date	The latest tool/service update date. In this case, only the open-source/prototype tool/services are considered when available.
	Number of Devices	Number	Number of available devices to execute the tests.
	Support Physical Device	Yes / No	Check if the tool/service supports the test on a physical device.
	Support Instrumented Tests	Yes / No	Check if the tool/service supports the execution of instrumented tests.
	Support Frameworks/Tools	Yes / No	Check if the tool/service supports different testing frameworks/tools to execute instrumented tests. For instance, JUnit, Appium, Espresso, etc.
Usability Feature	Project Configuration	Yes / No	Check whether is required to configure the project before executing the tests, e.g., if the user has to add an external dependency on the build file to operate the tool/service.
	External Configuration File	Yes / No	Whether is required to create an external file to execute the tests. There might be cases where creating an external configuration file to operate the tool/service is necessary.
	Format of Reports	Plain text	The formats of the reports generated by the tool/service. For instance, HTML, JSON, displayed on the web system, etc.
	Record Tests	Yes / No	Check if the tool/service supports the record of video/screenshot by default, without the need to add an external dependency. Android has an API that supports this feature. However, the user has to add this feature programmatically.
	Ease to Execute the Tests	Value ranging from 1 to 5	Assign a score based on the ease of operating the testing tool/service, i.e., evaluate the steps to execute the tests. For instance, analyze if the user needs to upload the *.apk* files on a server, if it is executed on the command line, how the devices are selected to execute the tests, etc. The score has been assigned based on the 5-Likert Scale (LIKERT, 1932).
Customization Feature	Attach External Devices	Yes / No	Check whether is possible to attach an external device. There might be cases where an application interacts with external devices.
	Attach External Test Case Generator	Yes / No	Whether it is possible to customize the tool/service by adding a test case generator.
	Attach External Test Criteria	Yes / No	Check whether customization of the tool/service is possible by attaching the test criteria approach or framework. For instance, if it is possible to use mutation testing to assess the quality of the tests.

and Spoon: (i) the developers currently maintain it, and (ii) it provides different features that may be useful for the company. Table 30 presents the selected tools/services for the comparative study.

Table 30 – Information of the selected tools/service for the experimental study.

Tool/Service	Developed/Supported by
AWS Device Farm (AMAZON, 2024)	Amazon
Firebase Test Lab (LLC, 2024)	Google LLC
Visual Studio App Center (CORPORATION, 2024)	Microsoft Corporation
Distributed Bug Buster (FARIA et al., 2019)	Faria et al.
Spoon (SQUARE, 2023)	Square
Marathon (MARATHONLABS, 2023)	MarathonLabs

4.4.5 App Selection

The app selection step consists of selecting a set of Android apps to be executed in each tool/service, aiming at gathering all the relevant information to help the decision process. Four apps of different categories available at F-Droid[6] have been selected. The only requirement for choosing the apps is that they should already have implemented instrumented tests.

The apps Cofi, dcnnt, PrepaidBalance, and TrailSense are available at F-Droid, whereas Von Braun Labs-app is the toy project we developed with a library created by Von Braun Labs to verify if the tools/services can run it. Table 31 presents detailed information about the applications, such as application name, application category, the programming language in which the app has been developed, target SDK, and the number of instrumented tests.

Table 31 – List of selected apps to execute the experimental study.

Name	Category	Language	Target SDK	# of Tests
Cofi	Time	Kotlin	33	42
dcnnt	Connectivity	Kotlin	33	4
PrepaidBalance	System	Kotlin	33	3
TrailSense	Navigation	Kotlin	33	5
Von Braun Labs-app	-	Java	32	13

4.4.6 Device Selection

Given the requirements, the comparative study only considered physical mobile devices. DBB, Spoon, and Marathon were installed on our local machine, and, in this case, we selected two Android devices that were available for testing: a Motorola G6 Play and a Samsung Galaxy A13.

On the other hand, AWS Device Farm, Firebase Test Lab, and Visual Studio App Center have a predefined set of physical devices. In this case, we tried to choose the same devices we had (i.e., Motorola G6 play and Samsung Galaxy A13) or those closely resembling both devices regarding model and characteristics. Table 32 presents the information about the devices.

[6] https://f-droid.org/

Table 32 – Information of the selected mobile devices.

Tool/Service	Devices	Android Version
Spoon / Marathon / DBB	Motorola G6 Play	9
	Samsung Galaxy A13	12
AWS Device Farm	Motorola G7 Play	9
	Samsung Galaxy A73	11
Firebase Test Lab	Google Pixel 3	9
	Google Pixel 6	12
Visual Studio App Center	Motorola G7 Play	9
	Samsung Galaxy A12	12

4.5 Comparative Study Results

As mentioned in the previous section, this experimental study compares the features of six testing tools/services to define the most suitable one for Von Braun Labs. We executed a set of five different applications to extract all the relevant features of each testing tool/service. Besides, the tests of each app were performed on two different physical mobile devices. In this case, we did not consider emulators because the application developed by Von Braun Labs has many features not supported by emulators. This section reports the results of the comparison.

As seen in Table 33, we filled the checklist according to the values defined in Table 29. Now, we detail each of the three main features (i.e., Technical Features, Usability Features, and Customization Features) and their specific features.

Table 33 – Result of comparison between the six tools/services.

Features	AWS Device Farm	Firebase Test Lab	Visual Studio App Center	Marathon	Spoon	DBB
Technical Features						
OS Supported	Android / iOS	Android / iOS	Android / iOS	Android / iOS	Android	Android
License	Commercial	Commercial	Commercial	Open-Source	Open-Source	Prototype
Last Release	-	-	-	March/2023	nov./2020	abr./2019
Number of Devices	110	68	634	-	-	-
Support Physical Devices	Yes	Yes	Yes	Yes	Yes	Yes
Support Instrumented Tests	Yes	Yes	Yes	Yes	Yes	Yes
Support Frameworks/Tools	Yes	Yes	Yes	Yes	Yes	Yes
Usability Features						
Project Configuration	No	No	Yes	No	No	Yes
External Configuration File	No	No	No	Yes	No	No
Format of Reports	Web System	Web System	Web System	HTML	HTML	Web System
Record Tests	Yes	Yes	Yes	Yes	No	No
Ease to Execute	3.75	3.75	2.75	4	4.25	2
Customization Feature						
Attach External Device	No	No	No	Yes	Yes	Yes
Attach External Test Case Generator	No	No	No	Yes	Yes	Yes
Attach External Test Criteria	No	No	No	Yes	Yes	Yes

4.5.1 Technical Features

OS Supported. Regarding the types of operating systems supported, AWS Device Farm, Firebase Test Lab, Visual Studio App Center, and Marathon offer support for both Android and iOS. On the other hand, Spoon and DBB only support Android devices.

License. There are three different types of license: *Commercial*, *Open-source*, and *Prototype*. Given the business model, AWS Device Farm, Firebase Test Lab, and Visual

Studio App Center are categorized as commercial. Both Spoon and Marathon are open-source tools. DBB is considered a prototype since it was developed to assess the feasibility of the approach defined by Faria et al. (2019).

Support Physical Devices. This feature is presented in all six tools/services.

Number of Devices. This feature presents the number of available Android devices for each tool/service. The number of devices of AWS Device Farm, Firebase Test Lab, and Visual Studio App Center is presented in their documentation. In this case, Visual Studio App Center has 634 available Android devices, followed by AWS Device Farm with 110 devices and Firebase Test Lab with 68 devices. On the other hand, the number of devices may vary considering the other three tools. In Marathon and Spoon, the number of available devices is the number of devices the user/company has to execute the tests. Considering DBB, the number of available devices depends on the devices registered in the web system (i.e., similar to a crowdsourced testing platform).

Support Instrumented Tests. Given the need to perform tests on a physical device, it is mandatory that the tool/service supports the execution of instrumented tests. All six tools/services support unit and UI-instrumented tests.

Support Frameworks/Tools. One crucial feature is the support of different tools and frameworks for test automation. Since most of the tests will be executed at the unit and UI level, the support of tools/frameworks is expected for this purpose. In this case, all six tools/services support different tools/frameworks for test automation, such as JUnit and Espresso.

4.5.2 Usability Features

This describes the usability features of each testing tool/service. In this case, “usability" relates to how we use the tool/service. Therefore, we aim to understand the steps required to execute each tool/service and how the results are presented to the user.

Project Configuration. This feature is related to configuring the project before executing it in the tool/service. In Visual Studio App Center, the user has to add a dependency to the build file and configure the test cases to generate the report. In DBB, the user has to add an external *.jar* file to execute the tests.

External Configuration File. This feature relates to the need for an external configuration file to execute the tool/service. Hence, only Marathon requires a script file with the execution configuration to perform distributed tests.

Format of Reports. One key feature of the tool/service is the presentation of a comprehensive report of the test execution. All six tools/services provide an easy-to-read report. However, they differ in the way it is presented. Marathon and Spoon locally export the report in *HTML* file. The other four services display the report information in their web system.

Record Tests. In this feature, we try to figure out whether the tool/service records the test execution by default, i.e., without configuring the project to record the tests. Spoon and DBB do not record the test execution by default.

Ease to Execute. This feature seeks to understand how easy is the execution of the tool/service. In this case, we defined a checklist based on the Likert's Scale (LIKERT, 1932) with four specific characteristics:

1. Tool/Service setup: this feature relates to the steps required to set up the tool/infrastructure for usage, i.e., tool/service installation, account creation, and payment information.

2. App preparation: this feature relates to the need to prepare the application before generating the *.apk* file, i.e., add external dependencies and configure test cases.

3. Device selection: this feature relates to how the devices are selected to perform the tests.

4. Test execution: this feature relates to the execution flow of the tool/service, i.e., the need to access a web system, the need to create a test plan, the need to upload the *.apk* files, and the need to create a configuration file to perform the tests.

A score ranging from 1 to 5 is assigned for each feature, where five means it is easily performed, and one means some difficulties in performing the feature. The final score is computed by the sum of each assigned score divided by 4.

AWS Device Farm received a total score of 3.75. The score assigned is three regarding *Tool/Service setup* criterion because the user has to create an account and provide billing information (i.e., credit card information). The score assigned in the *App Preparation* criterion is five since there is no need to prepare the app for execution, i.e., the user just has to build the application into an *.apk* file. Concerning *Device Selection* criterion, the devices used during the tests may be chosen in their web system; hence, the score assigned is 4. In *Test Execution* criterion, we focus on the steps required to execute AWS Device Farm. To run in default mode, the user has to access the web system, create a test plan, choose the most suitable devices for testing, and upload the *.apk* files. Given the effort to execute the tests, we assigned a score of 3.

Firebase Test Lab has also received a score of 3.75. In this case, the steps to execute this service are practically the same as AWS Device Farm. For instance, the user has to create an account and provide billing information, generate the *.apk* file without needing to add an external dependency and choose a set of available devices for testing. Moreover, the user has to access the web system, select the devices, and upload the *.apk* files.

The score assigned for Visual Studio App Center is 2.75. Regarding the execution flow, Visual Studio App Center has some similarities with the services mentioned above. However, some key differences may decrease the Visual Studio App Center score. The

score in *Tool/Service setup* criterion is two because the user needs to create an account and install an external dependency responsible for executing the tests and sending the reports back to the web system. The score assigned is 3 in *App preparation* criterion, given the need to add a dependency in the *build.gradle* file and prepare the test cases for report generation. The task of *Device selection* is similar to AWS Device Farm and Firebase Test Lab; hence, the score assigned is 4. Finally, in *Test Execution* criterion, the steps are similar to the other two services. However, in Visual Studio App Center, the user has to execute the tests using the CLI client previously installed.

Next, the tool Marathon received a final score of 4. To execute Marathon (*Tool/Service setup* criterion), the user has to install the binary file and also install Java and Android SDK packages (i.e., platform-tools). In this case, we assigned a score of 4. Concerning *App preparation* criterion, the score is 5 since the user must build the application without needing to import any external dependency. In the *Device selection* criterion, the devices used to execute the tests are those connected to the *adb* server. In this case, there is an effort to enable the *Development Options* and connect each device to the *adb* server. Thus, the score assigned is equal to 3. Considering *Test execution* criterion, the score is 4. In this case, the user has to configure a *YAML* file comprising all the information of the test execution and execute through CLI.

Overall, the execution flow of Marathon and Spoon are practically the same. The main difference between both tools is that Spoon does not require an external configuration file to execute the tests. Hence, Spoon has a final score of 4.25.

Finally, DBB has a lower final score of 2. A reasonable explanation for this score is that DBB has been developed as a prototype to show the feasibility of the author's approach (FARIA et al., 2019). Hence, the tool requires many improvements before a broader usage. Concerning the *Tool/Service setup criterion* criterion, the user has to configure the server properties and recompile the web system. Besides, the application responsible for the test execution should be compiled and installed on each device. Given the effort to set up the environment, we assigned a score of 1. The score is three in *App preparation* because the user has to add an external dependency to execute the tests. Regarding *Device selection* criterion, the score assigned is 3. In this case, the device selection is done on the web system. Finally, in *Test Execution*, the score is one since many tasks have to be manually performed. For instance, the user has to access the web system, create the test plan, upload the *.apk* files, and request a test execution. Moreover, the user has to access the mobile application, accept the test execution and install the application under test.

4.5.3 Customization Feature

With this feature, we aim to understand whether the tool/service supports adding new features to enhance the testing process at Von Braun Labs.

Attach External Device. The application of Von Braun Labs interacts with a specialized IoT device; hence, exploring the development of a controlled testing environment is interesting. Since the devices of AWS Device Farm, Firebase Test Lab, and Visual Studio App Center are remotely located, we cannot assure that it is possible to develop a controlled testing environment. On the other hand, in Marathon, Spoon, and DBB, we have total control of the device, making it possible to develop a testing environment where the devices interact with external IoT devices.

Attach External Test Case Generator. We expect to implement enhancements for the selected tool/service once the testing environment matures. One possible feature is adding a test case generator tool or framework. Since the source code of Spoon, Marathon, and DBB is available, it would be easier to implement this feature on these three tools.

Attach External Test Criteria. Similar to *Attach External Test Case Generator*, using different testing criteria to assess the quality of generated test cases is expected. Hence, it would be easier to implement this feature on Marathon, Spoon, and DBB.

4.5.4 Answer to Research Question

The comparative study provided an overview of the pros and cons of each tool/service and helped us in the decision process. It is important to emphasize that the analysis was carried out based on the scenario presented by the stakeholders.

Hence, we elected Marathon and Spoon as possible candidates for Von Braun Labs. In general, both tools have the same features. They are easy to execute, provide a comprehensive report, and it is possible to customize the tool to provide different features to enhance the testing process. The main difference is that the developers currently maintain Marathon.

After presenting and discussing the results with the stakeholders, they decided on **Marathon** in their testing environment.

4.6 Environment Setup – Case Study

4.6.1 Environment Setup

After selecting Marathon as the tool to support the execution of tests across multiple physical devices, we started to prepare the testing environment. This environment can be described as a local device farm with a server responsible for starting the execution of the tests and a set of different mobile devices attached to this server.

Initially, our job was to set up the environment according to the existing execution pipeline. Figure 36 summarizes the execution flow.

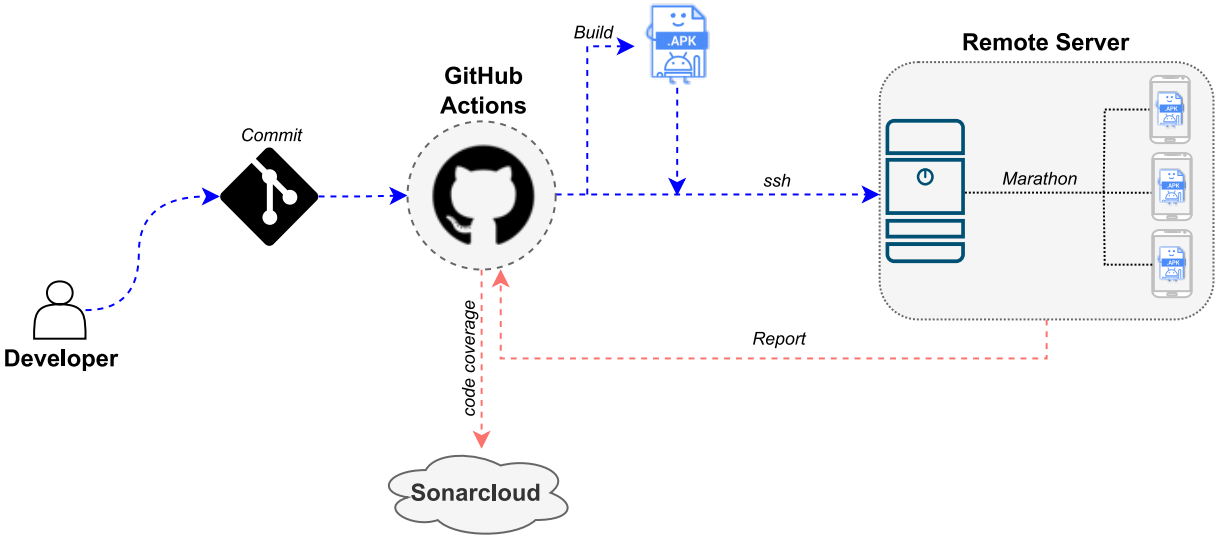


Figure 36 – Execution flow of the testing infrastructure.

Whenever the developers commit the code, GitHub Actions triggers a workflow to test the application using Marathon. The workflow starts by building and generating the *.apk* file containing the instrumented tests. Next, the *.apk* file is sent through *ssh* to a remote server where Marathon is deployed, and the devices are connected.

Finally, Marathon reports the test execution (i.e., provides information on the tests that passed or failed) and generates the *.ec* files (i.e., JaCoCo file containing information on code coverage). In the case of the test set quality, we decided to integrate the platform with SonarCloud, which is a quality gateway. SonarCloud consumes the coverage report provided by Marathon, and we can establish a quality gateway considering the desired level of coverage and application criticality.

In this first setup, we did not consider the interaction of the mobile devices with the external specialized IoT device of Von Braun Labs because we need to understand if the configured environment can execute their application's tests. Hence, we conducted a pilot study to validate this scenario.

To not compromise the CI/CD pipeline of Von Braun Labs, we set up a locally controlled environment to execute the pilot study. We used a personal computer to emulate the remote server and two mobile devices: (i) Motorola G6 Play and (ii) Samsung Galaxy A13.

4.6.2 Pilot Study

In the pilot study, we considered the application of Von Braun Labs to validate the testing environment. We cannot present detailed information about the application for intellectual property reasons. Instead, we will give an overview of this application.

Figure 37 summarizes the application. In summary, the application is responsible for communicating with the specialized IoT device located at different locations, collecting and preparing the received data, and sending it to a remote server. Then, the stored data is consumed by other applications of Von Braun Labs.

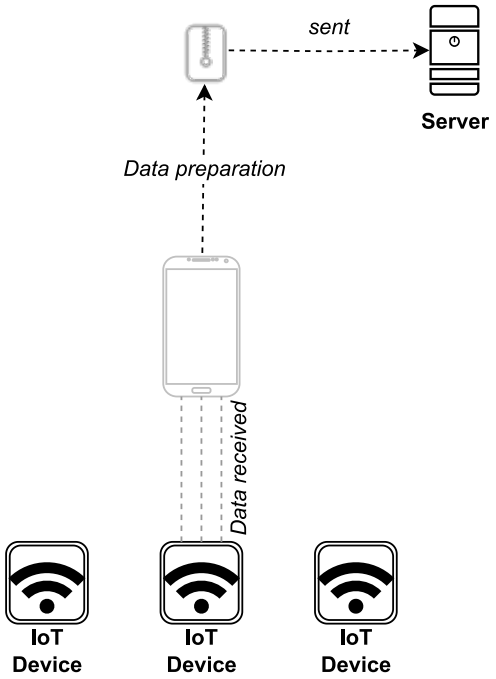


Figure 37 – Description of Von Braun Labs's application.

Regarding statistics, the application comprises 3.168 lines of code and 30 *.java* files. These files contain four interfaces, two enums, and 24 class files. In total, the application has 179 methods, considering both public and private methods.

The pilot study was carried out following three steps: (i) understanding the application, (ii) test case development, and (iii) test case execution.

4.6.3 Understanding the application

The first step involves getting the application's source code and understanding what the code should do. Despite the app not having a large code base compared to other industrial applications, some features related to the communication between the application and the sensors increase the app's complexity. In addition, there is no formal document that elicits the requirements, use cases, and architectural/technical features, making this task very challenging.

Therefore, we documented all the classes and methods in JavaDoc format. For instance, at the beginning of the project, there were 42 documented methods/classes. Now, we have improved this documentation detailing the 191 remaining classes/methods. The task of understanding the code and developing the test cases took three weeks to complete. In this case, we can conjecture that if there were an informal document containing

information about requirements and case uses, the execution time of the task would be significantly reduced. Moreover, the need to contact the developers to understand the application would also decrease.

4.6.4 Test Case Development

The next step consists of implementing some test cases to be executed in the defined infrastructure. We analyzed the existing test cases and found 7 unit test cases. These test cases are executed locally (i.e., do not execute on a mobile device) and have a code coverage of 10.8%, considering line and branch coverage.

We then started to analyze the source code, aiming at selecting the class to be tested. We derived the project into a dependency graph and decided to use a bottom-up approach, i.e., starting with those classes with few/no dependencies with other classes. In this case, it would be easier to isolate the module to be tested and track the behavior of the environment in case of an error.

Hence, we choose a class that checks the hardware properties of a given mobile device. This class has 47 lines of code and five public methods. We implemented seven unit-instrumented test cases; six were successfully executed, and one failed.

4.6.5 Test Case Execution

As described in Section 4.6.1, GitHub Actions is responsible for triggering the execution of Marathon at Von Braun Labs. First, it builds and generates the *.apk* files before sending them to the remote server. Next, it executes a bash script responsible for initializing the devices and starting the execution of Marathon.

Adding Marathon to the CI/CD pipeline increased the execution time of the workflow. However, this increase was already expected since the number of tasks in the workflow has also increased. If we only consider Marathon, the execution time had an average increase of 1 minute and 20 seconds. Note that we are considering the time spent to initialize the devices and the time spent to run the tests. Besides, the execution time of instrumented tests tends to be slower when compared with unit-local tests[7].

The code coverage increased from 10.8% to 12.5% considering the 14 test cases developed (7 unit-instrumented tests and seven unit-local test cases already implemented). We assumed code coverage because Von Braun Labs already uses this metric in other applications. It is important to emphasize that code coverage may not be a good indicator to assess the quality of a given test suite (INOZEMTSEVA; HOLMES, 2014). Hence, we intend to explore different testing criteria to enhance their testing process. For instance, provide an investigation on the use of mutation testing of mobile applications at Von

[7] https://developer.android.com/training/testing/instrumented-tests

Braun Labs, similar to some recent works that explore the use of mutation testing in the industry (ABDI; DEMEYER, 2022; BETKA; WAGNER, 2022; PETROVIć et al., 2018; PETROVIć; IVANKOVIć, 2018).

In general, the proposed testing environment worked as expected. We are aware that there are still many features to be implemented to optimize the execution of Marathon and also the need to execute the tests that communicate with the IoT devices. However, we believe that, as a starting point, the proposed environment will help Von Braun Labs to enhance the quality and reliability of their applications.

4.7 Lessons Learned and Contributions

4.7.1 Sometimes, less is more

We often tend to believe that a tool/service offering different features and functionalities is enough to solve all our problems. One may think that the usage of consolidated services such as AWS Device Farm (AMAZON, 2023), Firebase Test Lab (LLC, 2023), and Visual Studio App Center (CORPORATION, 2023) is enough to test their applications.

Considering Von Braun Labs, we observed that having a simple and flexible tool that supports different customization according to the stakeholder's needs was the best option. It may seem obvious, but in this case, it was essential to understand the scope of the applications to be tested and what the stakeholders expected to accomplish when defining the testing tool/service.

As a contribution, we provided a comparison framework to assess six available tools and services. We presented the characteristics and pros, and cons of each tool/service to support the stakeholder's decision process. Although the comparative study was applied only in the context of a single company, the comparison framework can be helpful to other companies with the same need, and, in addition, the framework can be adapted and extended to evaluate different technologies.

4.7.2 The importance of having a testing process since the beginning of the development project

One of this study's challenges is defining the testing process in an ongoing project since the application is already in production. In this case, we had to understand the current scenario to determine how the tests should be conducted and which features to prioritize.

As a contribution, we analyzed and grouped the classes to be tested according to the types of tests: unit-local test, unit-instrumented test, and UI-instrumented test. Figure 38 summarizes the clusterization.

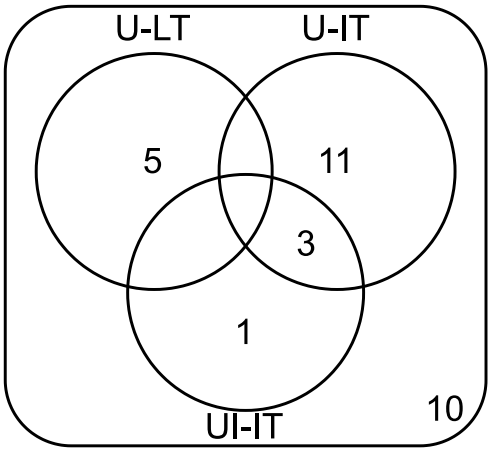


Figure 38 – Clusterization of the tests according to their types.

The set $U - LT$ relates to unit-local tests; hence, we analyzed that we will only develop unit-local tests in five classes. In $U - IT$, we grouped eleven classes in which we will develop unit-instrumented tests. And the group $UI - IT$ (UI-instrumented tests) has only one class. In $(U - IT \cap UI - IT)$, there are three classes where we will perform both unit-instrumented tests and UI-instrumented tests. Finally, there are ten classes where we will not develop test cases because they are just simple enum and entity declaration classes.

With this categorization, we can save execution resources because there are tests that do not need to be executed on mobile devices, i.e., the tests are run during build time. Moreover, it helped prioritize the features to be tested and define an incremental approach to design the tests. For instance, we can start with a bottom-up approach by developing unit-local tests and unit-instrumented tests of the classes with few/no dependencies. Then, we can evolve the tests to those features with more dependencies until we reach UI-instrumented tests.

Although applying testing activities from the beginning of the project is straightforward and well-known both in academia and the industry, we observed that in the mobile ecosystem, there is still difficulty in applying tests. For instance, Pecorelli et al. (2020) conducted an empirical study in which 1,780 Android applications were analyzed, and the results showed that less than 50% of the applications have tests. Observe that the results reflect the scenario of Von Braun Labs and, in this case, there is a need to understand why testing is still not widely applied when considering mobile applications. Thus, we are conducting a questionnaire survey with the testers and developers of Von Braun Labs to highlight the difficulties in implementing tests considering an industrial scenario.

4.7.3 The importance of academia-industry contribution

Different studies stated the need for contributions between academia and industry (TRAMONTANA et al., 2019; JúNIOR et al., 2021). This ongoing project with Von Braun Labs may provide some insightful ideas on how to lessen these gaps.

Heretofore, the collaboration with Von Braun Labs has many positive points. Regarding communication, the stakeholders are very open-minded to our ideas and contributions to enhance the testing process. Another positive point is that the stakeholders provide all the resources needed to conduct our study (e.g., the source code of the applications and grant access to the technologies used in their environment). In this case, we can provide experimental studies to validate different research proposals in an industrial setting. For instance, we can assess the validity of mutation testing tools using an industrial Android application.

Furthermore, this academia-industry partnership can benefit other companies with the same difficulty carrying out tests that interact with external devices. The results can guide these companies in defining an environment to run tests on multiple devices that can either use simulators to emulate an external device or create a real environment in which the external devices are physically allocated.

4.8 Related Work

Several studies in academic literature compare tools, frameworks, techniques, and approaches. The recent study of Amalfitano et al. (2022) compares different Java mutation testing tools according to their features considering five dimensions: Version, Deployment, Mutation process, User-centric features, and Mutation Operators. Our work compares different tools/services that support the execution of instrumented tests across multiple devices. In this case, our comparison framework and tool/service selection was defined based on the need of the stakeholders. Considering mobile application services, the work of Starov et al. (2015) compared 13 device clouds based on supported platforms (Android, iOS, Other), types of testing, and delivery types (public or private).

The study of Arif e Ali (2019) compared seven automated testing tools for mobile applications. In this case, the authors compared these tools based on the supported platforms, the benefits and limitations of each tool, and the pricing plan. Regarding the Android ecosystem, the studies of Choudhary, Gorla e Orso (2015) and Wang et al. (2018) compared different test generation tools. Choudhary, Gorla e Orso (2015) analyzed seven test input generation tools based on four criteria: ease of use, Android framework compatibility, code coverage achieved, and fault detection ability. The study of Wang et al. (2018) also compared different test generation tools concerning code coverage and fault detection ability. The main difference between both studies is that Wang et al. (2018) considered 68 industrial apps to perform their analysis.

Many studies have proposed different ways to perform mobile application testing across multiple devices. Malini et al. (2014) proposed an MTaaS framework deployed on the cloud to perform multiple-device tests. This framework considers three types of testing: load testing, performance testing, and functional testing. Tao e Gao (2017) propose the development of an Infrastructure-as-a-Service (IaaS) system for mobile TaaS considering multiple devices and emulators. Mozgovoy e Pyshkin (2018) developed a mobile farm infrastructure to execute UI testing focusing on non-native mobile applications.

Finally, Faria et al. (2019) provided a disruptive platform (DBB) based on the concept of collaborative economy to perform instrumented tests on multiple devices. The main feature of the proposed platform is the possibility to perform the tests without the need to have the device connected to a USB port at a low-cost model. In our study, we considered DBB in the tool/service comparison.

4.9 Conclusion and Future Works

The present study reports a practical experience of implementing a testing infrastructure to support testing in multiple Android devices to enhance the testing process of Von Braun Labs. In this scenario, the stakeholders stated (i) the need to perform tests on physical devices and (ii) the tests must interact with specialized IoT devices.

To define the most suitable tool, we propose a comparative study to assess the features of 6 different tools/services: AWS Device Farm, Firebase Test Lab, Visual Studio App Center, DBB, Spoon, and Marathon. As a main contribution, we propose a comparison framework based on three dimensions: Technical Features, Usability Features, and Customization Features. The comparison framework provides the basis to support the decision-making process of the most suitable tool/service for Von Braun Labs. Observe that the comparison framework can be helpful to other companies with the same scenario, and, in addition, the framework can be adapted and extended to evaluate different technologies.

We execute five Android apps in each tool/service and consider two different Android devices to collect all the relevant data. After presenting the results, the stakeholders decided on Marathon as the tool for Von Braun Labs. Next, we conduct a pilot study to validate the infrastructure using an application of Von Braun Labs. In this case, we had to configure the execution of Marathon in their CI/CD pipeline.

Moreover, we analyzed the current state of the application and started the proposition of improvements for their testing process. We grouped the features to be tested according to the types of tests: unit-local test, unit-instrumented test, and UI-instrumented test. Besides, we proposed a bottom-up testing approach where we develop unit-local tests and

unit-instrumented tests of the features with few/no dependencies. Then, the strategy evolves to those features with more dependencies until UI-instrumented tests. Finally, we discuss the lessons learned and contributions of the partnership with Von Braun Labs.

Regarding future works, we propose the following:

- ❏ Survey with the developers/testers of Von Braun Labs to understand how the testing infrastructure impacted the quality of the application.
- ❏ Analyze the number of bugs found after implementing the testing infrastructure. In this case, we expect to understand how the commits related to bug fixes evolved after implementing the testing infrastructure.
- ❏ Analyze different testing criteria considering the applications of Von Braun Labs.
- ❏ Explore test case generation using LLM considering the applications of Von Braun Labs.
- ❏ Conduct more case studies considering the interaction with the specialized IoT devices.

Chapter 5

Designing Mutation Operators for Android Device Components: A View Through Bluetooth and Location API's

5.1 Overview

This chapter presents the fourth published paper during the PhD. The next stage of the PhD project began after defining the local device farm presented in the previous chapter. In this stage, the project pivoted to assess mechanisms for evaluating existing tests and to investigate strategies for generating new test cases automatically. Given the application domain, we decided to examine the adoption of mutation testing for its ability to model faults, its consideration of different scenarios, and its recognized capabilities for estimating the quality of a given test suite.

Given the partner company's application characteristics, the focus was on investigating how to apply mutation testing to the Android API's hardware components for location or beacon detection, namely, GPS and Bluetooth sensors. Therefore, this paper proposes a set of mutation operators that consider these specific components. Below is the information regarding the paper (see also Figure 39):

- ❑ **Title:** Designing Mutation Operators for Android Device Components: A View Through Bluetooth and Location APIs.

- **Authors:** Pedro Henrique Kuroishi (UFSCar), Ana Cristina Ramada Paiva (Universidade do Porto), José Carlos Maldonado (ICMC-USP), and Auri Marcelo Rizzo Vincenzi (UFSCar).
- **Local:** XXXIX Simpósio Brasileiro de Engenharia de Software.
- **Year:** 2025
- **Status:** Published.
- **DOI:**<https://doi.org/10.5753/sbes.2025.9878>

Designing Mutation Operators for Android Device Components: A View Through Bluetooth and Location API's

Pedro Henrique Kuroishi
Department of Computing
Federal University of São Carlos
São Carlos, Brazil
phk@ufscar.br

Ana C. R. Paiva
INESC TEC, Faculty of Engineering, University of Porto
Porto, Portugal
apaiva@fe.up.pt

José Carlos Maldonado
Institute of Mathematics and Computer Science
University of São Paulo
São Carlos, Brazil
jcmaldon@icmc.usp.br

Auri Marcelo Rizzo Vincenzi
Department of Computing
Federal University of São Carlos
São Carlos, Brazil
auri@ufscar.br

Figure 39 – Information regarding paper P4.

5.2 Abstract

Context: Mutation operators play a crucial role during the mutation testing process due to their capability to model common faults to be injected into an application under test. In the context of the Android OS, the academic literature shows that different studies propose mutation operators, especially focusing on the GUI and configuration. On the other hand, few devise mutation operators for specific Android device resource components such as connectivity, location, and sensors. **Objective:** Therefore, this paper aims to investigate the design and proposition of mutation operators for Bluetooth, Location, and the third-party library AltBeacon. The rationale is that this paper is carried out in an academia-industry partnership, in which the company develops applications that rely on these Android components. **Method**: The design process used a systematic approach named HAZOP to minimize any possible bias. **Results**: This systematic process helped in deriving a set of 16 mutation operators. Next, the paper provides an empirical cost evaluation that assesses the number of generated mutants for two applications and

the number of generated mutants per operator, showing the feasibility of the mutation operators and their capabilities in modeling real faults. **Conclusion**: Finally, the paper discusses the future perspectives of extending the operators to other device resource components such as Wi-Fi, NFC, and sensors, as well as automation perspectives by envisioning the implementation and validation of the mutation operators.

5.3 Introduction

Mutation testing (or mutation) is a test criterion commonly adopted to assess the quality of a given test suite based on its fault detection capabilities (DEMILLO; LIPTON; SAYWARD, 1978; PAPADAKIS et al., 2019). The traditional mutation testing process starts with the generation of faulty versions, named *mutants*, of the original application under test (or AUT). Then, the test suite is executed against each mutant, and the output is evaluated to check whether the test suite can detect the fault injected in the mutant. The quality metric is given by the mutation score, which is defined by the ratio of killed mutants to non-equivalent ones. Observe that a crucial step of mutation testing consists of generating the faulty version of the AUT. To support this process, mutation testing relies on *mutation operators*, i.e., a set of rules that modify the original application by making a simple syntactic change (JIA; HARMAN, 2011). For example, an arithmetic operator changes the statement $a = b + c$ to $a = b - c$ and $a = b * c$.

Over the years, many studies have proposed mutation operators for different programming languages (e.g., C (AGRAWAL et al., 1989; DELAMARO; MALDONADO; MATHUR, 2001), Java (KIM; CLARK; MCDERMID, 2000b; KIM; CLARK; MCDERMID, 2000a; MA; KWON; OFFUTT, 2002; DELAMARO; PEZZè; VINCENZI, 2001; BRADBURY; CORDY; DINGEL, 2006; MA; OFFUTT; KWON, 2006), Python (DEREZIńSKA; HAłAS, 2014), JavaScript (MIRSHOKRAIE; MESBAH; PATTABIRAMAN, 2013; NISHIURA et al., 2013; MUZAMAL; NADEEM, 2019)) and system types (e.g., Android (DENG; OFFUTT; SAMUDIO, 2017; DENG et al., 2017; JABBARVAND; MALEK, 2017; LUNA; ARISS, 2018; PAIVA et al., 2019; LIU et al., 2020; ESCOBAR-VELáSQUEZ et al., 2022), deep learning systems (MA et al., 2018), time systems (VEGA et al., 2018), cyber-physical systems (VIGANò et al., 2023)). This paper focuses on designing mutation operators for Android.

Android application testing differs from traditional software (e.g., web and desktop applications). An Android app is designed to run on a mobile device. Therefore, when testing an Android app, one has to consider the limitations of the mobile device, such as screen size and density, multiple sensors and connectivity, and constrained resources (MUCCINI; FRANCESCO; ESPOSITO, 2012). Given these heterogeneous characteristics, it is expected that the mutation operators will cover not only the functionality of the app but also other aspects of the Android ecosystem.

The mapping study of Silva et al. (2022) showed that few studies focus on mutation testing for Android apps. Consequently, only a subset of the mapped studies proposed Android mutation operators focusing on adapting existing Java operators and designing specific operators for GUI components. Additionally, only four studies (JABBARVAND; MALEK, 2017; DENG et al., 2017; LIU et al., 2020; ESCOBAR-VELáSQUEZ et al., 2022) present mutation operators for other Android components such as sensors, connectivity, and location.

The present paper aims to propose the design of mutation operators for Bluetooth (LLC., 2025a) and Location (LLC., 2025b) API, given the lack of studies focusing on these specific Android components. For Bluetooth, this paper considers both the classic and low-energy APIs. Moreover, the present study explores the design of mutation operators for the third-party library AltBeacon (NETWORK; YOUNG, 2025). For Location, the operators are designed for the native and the Fused Location Provider API provided by Google Play Services (or GMS) (LLC., 2025c).

The rationale for selecting these specific components is that the study is being carried out in an academia-industry partnership. In this case, the company develops different applications that rely on these Android components. Therefore, the mutation operators provided in this study are not only useful in enhancing the quality of the apps developed by the company, but also benefit further academic research and companies dealing with the same difficulties.

In general, the design of mutation operators relies on the experience and knowledge of the proponent (KIM; CLARK; MCDERMID, 2000b). Since there is still no standardized method to facilitate the generation of operators, the paper leverages a systematic technique called Hazard and Operability Studies (or HAZOP) (KIM; CLARK; MCDERMID, 2000b; CHUDLEIGH et al., 1995). This approach originated in the chemical industry, and the goal is to identify potential hazards and deviations in the behavior of a system using a set of predefined guide words to support determining the cause and consequences of the deviations (CHUDLEIGH et al., 1995; KIM; CLARK; MCDERMID, 2000b; ARAUJO et al., 2011). The rationale for using HAZOP is to facilitate the design process and minimize any possible bias that may arise.

In summary, this paper makes the following contributions:

- ❏ The design of 16 mutation operators for Bluetooth, AltBeacon, and Location API's.
- ❏ A cost evaluation of the proposed mutation operators.
- ❏ A discussion of future perspectives on extending the defined set of mutation operators for other Android components and automation aspects.

The remainder of the paper is organized as follows: Section 5.4 presents an overview of the concepts of this paper. Section 5.5 describes the study goals and design. Section 5.6

provides the steps carried out to devise the mutation operators and a description and examples of them. Section 5.7 presents the empirical cost evaluation of applying the mutation operators. Section 5.8 describes the related works. Finally, Section 5.9 presents future perspectives and concludes the paper.

5.4 Background

This section provides a brief overview of the main concepts related to the present work.

5.4.1 Mutation Operators for Android

Mutation operators play a crucial role during the mutation process due to their capability of introducing a syntactic modification under the AUT (AGRAWAL et al., 1989). The academic literature proposes various mutation operators that can be applied to different programming languages, systems, and other purposes.

As mentioned, mutation operators can be described as a transformation rule that injects a possible fault in the source code, mimicking the common mistakes of developers (JIA; HARMAN, 2011). In general, these operators can modify variables, replace operators from expressions, delete statements, and so on.

In the context of Android, there are still few studies focusing on mutation testing and, consequently, there is still a need to explore this technique and the extension of the existing mutation operators (SILVA et al., 2022). Linares-Vásquez et al. (2017) proposed an insightful taxonomy of Android bugs to support designing the mutation operators. The authors categorized the existing bugs into fourteen categories: Activities and Intent, Back-end Services, Collections and Strings, Data/Objects Parsing and Format, Threading, Android Programming, Non-functional Requirements, GUI, I/O, Device/Emulator, API and Libraries, Connectivity, Database, and General Programming. Moreover, the authors stated that the bugs affect not only Android apps but also Java applications.

According to Silva et al. (2022), most Android mutation operators are centered on the GUI and the specific configuration of the Android API. On the other hand, few studies explore mutation operators for sensor and location. Additionally, few studies investigated the design of mutation operators for the Bluetooth API, considering connectivity.

Given the complexity of the ecosystem and the need to explore other Android components, we advocate for a broader investigation of designing mutation operators for Bluetooth and Location apps.

5.4.2 Bluetooth and Location API

Many Android application relies on the information retrieved by the sensors embedded in the mobile device. For instance, an app may use Bluetooth to connect and exchange data with a peripheral or an IoT device. Or may collect real-time information about the user's location or monitor the motion of a mobile device. Observe that these types of applications resemble the ones developed by our partner company.

According to the documentation (LLC., 2025a), Android API supports two types of Bluetooth API: classic and low-energy (BLE). The main difference between these two types relates to energy consumption. The first is commonly used for more battery-intensive operations, whereas the latter is projected to consume less energy and transfer a low quantity of data. The two types of Bluetooth may perform the same operations: finding devices, connecting, and transferring data.

BLE can be used in various contexts such as proximity-based applications, communicating with IoT systems, and interacting with BLE devices such as beacons (FüRST et al., 2018). The third-party Android Beacon Library (NETWORK; YOUNG, 2025) supports functionalities to facilitate the interaction of an Android device and beacons. It is worth noting that this library implements the AltBeacon advertisement protocol and is adopted by many applications (NETWORK; YOUNG, 2025; GOWRISHANKAR; MADHU; BASAVARAJU, 2015; LAKSHMI et al., 2019).

Android API also offers support to create location-aware applications (LLC., 2025b). Similar to Bluetooth, the Location API offers a ton of functionalities that facilitate collecting and processing information about the location. A developer may leverage the native functionalities and also the API supported by Google Play services (or GMS) to build location-aware apps. In this paper, the native and the GMS are considered.

5.4.3 Hazard and Operability Studies – HAZOP

The process for designing and proposing mutation operators is commonly carried out based on previous experience and knowledge (KIM; CLARK; MCDERMID, 2000a) since there is still no standardized methodology to facilitate this task. Kim, Clark e Mcdermid (2000a) designed mutation operators for the Java programming language using a systematic approach named Hazard and Operability studies (or HAZOP). This method originated in the chemical industry to identify potential hazards and deviations in the behavior of a system using a set of predefined guide words that have to be adapted and interpreted according to the system context (CHUDLEIGH et al., 1995; KIM; CLARK; MCDERMID, 2000b; ARAUJO et al., 2011). In general, the following guide words are considered: *no/none*, *more*, *less*, *as well as*, *part of*, *reverse*, and *other than*. Table 34 describes the interpretations of the guide words.

Table 34 – Guide words interpretation adapted from (KIM; CLARK; MCDERMID, 2000a; CHUDLEIGH et al., 1995).

Guide word	**Interpretation**
no/none	No part of the intention is achieved
more	Quantitative increase
less	Quantitative decrease
as well as	The design of the intent is achieved but with additional results
part of	Only some part of the intention is achieved
reverse	Reverse the information flow
other than	A result other than the original intent is achieved

Additionally, Chudleigh et al. (1995) extends this list with four guide words: *early*, *late*, *before*, and *after*. For the present paper, we considered the seven guide words from Table 34 and also *early* (i.e., earlier execution than intended) and *late* (i.e., delayed execution than intended).

For example, Kim, Clark e Mcdermid (2000b) applied the HAZOP approach to derive Java mutation operators by examining the language specification and identifying plausible deviations and their causes and consequences. The operator *Variable replacement operator* can be associated with the guide word *Other than*. It can replace variable names of the same type (*cause*), leading to an anomalous behavior of the system (*consequence*).

5.5 Study Setup

5.5.1 Study Goal

The main purpose of this paper is to investigate and design a mutation operator for the Android Bluetooth and Location API. The rationale for selecting these two components is the following:

1. As presented by Silva et al. (2022), few studies focus on mutation operators for specific Android components rather than Java-specific or GUI. Therefore, we intend to provide an investigation toward the viability of proposing mutation operators for Bluetooth and Location apps and envision the possibility for other connectivity components (e.g., Wi-Fi[1] and NFC[2]) and sensors[3] (e.g., gyroscope and accelerometer).

2. The present study is carried out under an academic-industry collaboration. The company develops mobile applications that interact with IoT devices using Bluetooth (both native and AltBeacon) and Location features. Hence, this paper focuses

[1] https://developer.android.com/develop/connectivity/wifi/wifiscan
[2] https://developer.android.com/develop/connectivity/nfc
[3] https://developer.android.com/develop/sensors-and-location/sensors/sensors_overview

on these two components and aims to enhance the quality of the testing process of the company. For confidentiality reasons, this paper will not provide any information about the company or specific details of their applications.

5.5.2 Study Design

Figure 40 illustrates the steps carried out to guide the present study.

The process started by evaluating the Bluetooth, Location, and AltBeacon API and identifying possible mutation points. Then, the collected mutation points were assessed and grouped by their actions to form an initial subset of mutation operators. The subset was then validated using the HAZOP approach. Next, the operators were cross-checked against the existing mutation operators from the academic literature to verify if the operators were previously proposed. Whenever a mutation operator from the literature was not considered in our analysis, it was added to the final set of operators.

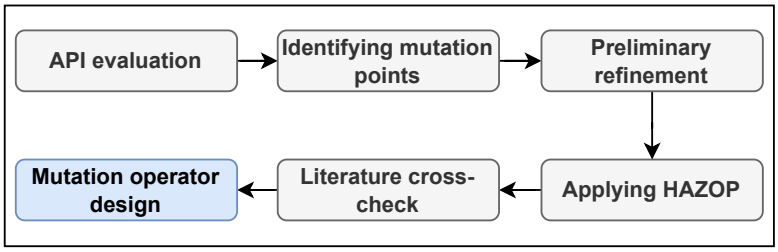


Figure 40 – Mutation operator designing process.

5.5.3 Empirical Cost Evaluation

After defining the mutation operators, this paper evaluates the cost of applying the operators. To this purpose, two apps were considered, and the mutants were manually generated to collect information about (i) the number of mutants generated per application and (ii) the number of mutants generated per operator.

This data provides insights into the viability of the mutation operators in terms of mutant generation and helps us to understand the rationale of the operators that eventually do not generate any mutants.

It is worth noting that the mutants were manually generated because it is still not implemented. That is, whenever a mutation point was found, the source code was manually modified to mimic a possible mutant, and the application was recompiled to check the validity of the syntactic change. Section 5.9 provides a broad discussion of the automation aspects.

5.6 Mutation Operator Description

The present section presents the design process carried out to devise the set of mutation operators. Additionally, the section provides a broad description of the operators and examples to enlighten their usage.

5.6.1 Designing Process

The first step of the designing process started with the API evaluation as displayed in Figure 40. It encompasses an in-depth Bluetooth, Location, and AltBeacon API documentation analysis to understand whether it was reasonable to apply mutation testing in these specific Android components.

During the analysis, it was possible to observe that some mutation points could be categorized based on the component action target. Android offers two types of Bluetooth API support: classic and low-energy. In both cases, the mutation points could be related to configuration, scan, connect/pair, and data/information transfer. Altbeacon also has four action targets: configuration, transmitting, ranging, and monitoring. Location can be divided into two types: native API and GMS (i.e., API that is part of Google Play Services). In both cases, the mutation points could be related to configuration and location-specific methods.

The next step consisted of identifying potential mutation points for each component action target. Table 35 presents the number of identified mutation points.

Table 35 – Number of identified mutation points.

API	Type	Action	# MP
Bluetooth	Classic	Configuration	5
		Scan	8
		Connect/Pair	4
		Transfer	1
	BLE	Configuration	1
		Scan	3
		Connect/Pair	3
		Transfer	1
	AltBeacon	Configuration	16
		Transmitting	5
		Ranging	7
		Monitoring	5
Location	Native	Configuration	7
		Location-specific	9
	GMS	Configuration	4
		Location-specific	9
Total			88

As can be seen, the previous step generated a larger number of possible mutation points. Moreover, various mutation points had the same code structure, that is, they

could represent the same type of mutation operator. Therefore, a refinement step was carried out to aggregate the mutation points and form a more fine-grained group type. After analysis, the 88 mutation points were clustered into 16 groups, which represent the preliminary set of mutation operators.

The name of each mutation operator represents the modeled faults. In a first attempt, we envisioned using the keyword *Sensor* to represent this type of mutation operator. However, Android offers support for Sensor API that does not directly correlate to Bluetooth and Location and hence, using this keyword would be misleading. Since both cases rely on mobile device resources, we decided to use the keyword *Device* to make a more generic name. Note that we intend to extend the mutation operators, considering other mobile device hardware components such as NFC, Wi-Fi, and sensors (e.g., gyroscope and accelerometer). Thus, it would be easier to aggregate new operators using a generic classification. Table 36 presents the preliminary set of mutation operators.

Table 36 – Set of devised mutation operators.

Operator Name	**Acronym**
BuggyDeviceCallback	BDC
TrapDeviceCallback	TDC
BuggyDeviceListener	BDL
TrapDeviceListener	TDL
NullDeviceInstanceDeclaration	NDID
DefaultDeviceBuilderInstanceDeclaration	DDBID
SwitchConditionalDeviceMethod	SCDM
ReplaceConditionalParameterDeviceMethod	RCPDM
ShiftDeviceMethodCall	SDMC
DeletionDeviceMethods	DDM
DeviceVariablesOperator	DVO
NullReferenceDeviceMethods	NRDM
RandomActionIntentDeviceDefinition	RAIDD
ReplaceCompatibleTypeDeviceGetMethods	RCTDGM
RandomLocationProviderDeviceReplacement	RLPDR
RandomLocationRequestBuilderPriorityDeviceReplacement	RLRBPDR

Table 37 – Results of HAZOP.

Acronym	**Guide Word**	**Cause**	**Consequence**
BDC	OTHER THAN	Changes the current instantiation of a callback method with a null value.	Possible error, or the application may throw a NullPointerException.
TDC	AS WELL AS	Add a trap method to check the reachability of a callback method.	No loss of information, i.e., the callback method is reachable.
BDL	OTHER THAN	Changes the current instantiation of a listener method with a null value.	Possible error, or the application may throw a NullPointerException.
TDL	AS WELL AS	Adds a trap method to check the reachability of a listener method.	No loss of information, i.e., the listener method is reachable.
NDID	OTHER THAN	Changes the current instantiation of a method with a null value.	Possible error, or the application may throw a NullPointerException.
DDBID	OTHER THAN	Changes the current instantiation of a method with a default instance.	Possible error or a loss of object information.
SCDM	REVERSE	Reverts the current return value of a boolean method.	Unexpected result or behavior of the application.
SCPDM	REVERSE	Reverts the current boolean parameter of a method.	Unexpected result or behavior of the application.
SDMC	EARLY/LATE	Makes an early/late call of a method.	Possible application error.
DDM	NO/NONE	Deletes a declared method.	Unexpected behavior or collateral effects, such as increasing energy consumption.
DVO	MORE/LESS	Increases/Decreases a numeric value.	Possible unexpected result.
NRDM	OTHER THAN	Changes the reference parameter of a method with a null value.	Possible error or an unexpected result/behavior of the application
RAIDD	OTHER THAN	Randomly changes the value of a declared action intent.	Unexpected result or behavior of the application.
RCTDGM	OTHER THAN	Changes the declaration of a get method with a similar with the same return type	Possible application error.
RLPDR	OTHER THAN	Changes the provider of a Location method.	Unexpected behavior or collateral effects, such as increasing energy consumption.
RLRBPDR	OTHER THAN	Changes the priority of a LocationBuilder instantiation.	Unexpected behavior or collateral effects, such as increasing energy consumption.

Next, the HAZOP approach was applied to validate the preliminary set of operators. Each operator was assigned to one or more guide words. An operator was excluded from the set whenever the association between the operator and the guide word did not occur.

Additionally, we describe the possible cause (i.e., the deviation/mutation operator that should be applied) and the consequence (i.e., system/code behavior when the operator is applied). Table 37 summarizes the results of applying HAZOP.

Nine mutation operators were associated with *OTHER THAN* guide word, which consists of replacing the current instantiation, properties, constants, and method call, causing unexpected behavior or collateral effects. Two mutation operators associated with *AS WELL AS*, that is, it inserts a code snippet without causing any loss of information. Two operators associated with *REVERSE* may cause an expected result from the application by switching the boolean value of a method or a method parameter. One operator associated with *EARLY* and *LATE* causes an application error. One operator is associated with *MORE* and *LESS*, which increase/decrease the value of a scalar variable, causing an unexpected result. Finally, one operator associated with *NO/NONE* leads to collateral effects. After analysis, no mutation operators were left without an association with the guide word. Thus, the final set of mutation operators is those presented in Table 36.

In the last step, the 16 mutation operators were cross-checked with the existing operators proposed in previous work. According to Silva et al. (2022), four studies proposed mutation operators for these Android components. Jabbarvand e Malek (2017) designed eight mutation operators for Location and four for Bluetooth. Linares-Vásquez et al. (2017) proposed one mutation operator for Location and two for Bluetooth. Deng, Offutt e Samudio (2017) contributed with one Location operator and Liu et al. (2020) designed one operator for Location and one for Bluetooth. Table 38 presents the relationship of the mutation operators from literature (see column *Literature Operators*) and the ones devised in the present paper (see column *Related to*).

Table 38 – Mutation operator subsumption.

Ref	**Literature Operators**	**Related to**
(JABBARVAND; MALEK, 2017)	LUF_T, LUF_D	DVO
	LRP_C, LRP_A	RLPDR
	RLU, RLU_P, RLU_D	DDM
	LKL	RCTDGM
	UAB	SCDM
	FDB_H, FDB_S	DVO
	RBD	DDM
(LINARES-VáSQUEZ et al., 2017)	NullGPSLocation	NDID
	BluetoothAdapterAlwaysEnabled	SCDM
	NullBluetoothAdapter	NDID
(DENG; OFFUTT; SAMUDIO, 2017)	LCM	RCTDGM
(LIU et al., 2020)	NullLocation	NDID
	NullBluetoothAdapter	NDID

The characteristics of the 18 mutation operators proposed in the literature were compared with the 16 operators of this study. An operator subsumes another if both have the

same characteristic. For instance, the mutation operators NullGPSLocation and NullBluetoothAdapter from (LINARES-VáSQUEZ et al., 2017) have the same characteristics as NDID. Therefore, NDID subsumes both operators.

After analysis, the mutation operators presented in this paper subsume all the operators proposed in the previous study, showing the feasibility and generalizability of our operators.

5.6.2 Mutation Operator Description

Table 39 summarizes information on the proposed mutation operators.

Table 39 – Mutation operator categorization.

Category	Mutation Operator
Replacement	DDBID
	DVO
	RAIDD
	RCTDGM
	RLPDR
	RLRBPDR
Null	BDC
	BDL
	NDID
	NRDM
Switch	SCDM
	SCPDM
Trap	TDC
	TDL
Deletion	DDM
Shift	SDMC

As can be observed, six mutation operators are categorized as *Replacement* (DDBID, DVO, RAIDD, RCTDGM, RLPDR, and RLRBPDR). Four mutation operators are categorized as *Null* (BDC, BDL, NDID, NRDM). Two are grouped as *Switch* (SCDM and SCPDM). Two operators are categorized as *Trap* (TDC and TDL). DDM is categorized as *Deletion* and SDMC as *Shift*.

The mutation operator marked with *L* refers to those that subsume an operator previously proposed in the academic literature. A brief description of the relationship between the operators is also provided.

5.6.2.1 Replacement Operators

This category groups six different mutation operators: DDBID, DVO, RAIDD, RCTDGM, RLPDR, and RLRBPDR. The main goal of this type of operator is to replace the current value of variables, method calls, parameters, and constants, or to instanti-

ate an object with a different value instead of null. A brief description of the operators is presented below, and Figure 41 illustrates an example of applying three replacement mutation operators.

- **DefaultDeviceBuilderInstanceDeclaration (DDBID)**: The operator replaces the current instantiation of a Location or Bluetooth class with a default implementation. These classes can be instantiated with different parameters and/or properties. Therefore, this operator declares a builder instance without these parameters/properties.
- **DeviceVariablesOperator (DVO) – L**: The operator mutates scalar variables. In this case, the scalar variable may have its value incremented and decremented. It can be associated with operators LUF and FDB designed by Jabbarvand e Malek (2017).
- **RandomActionIntentDeviceDefinition (RAIDD)**: The operator replaces the current declaration of a Bluetooth or Location action intent instance with a random value. It is worth noting that the action intent is mutated with another one of the same class.
- **ReplaceCompatibleTypeDeviceGetMethods (RCTDGM) – L**: The operator replaces the call of a get method of a given return type with another get method of the same return type and attached to the same object. This operator can be associated with the operator LKL from Jabbarvand e Malek (2017) and LCM from Deng, Offutt e Samudio (2017).
- **RandomLocationProviderDeviceReplacement (RLPDR) – L**: The operator replaces the Provider constant parameter of a Location instance with a custom String value or an existing LocationManager provider. Observe that this operator is specific to the Location and can be associated with operator LRP from Jabbarvand e Malek (2017).
- **RandomLocationRequestBuilderPriorityDeviceReplacement (RLRBPDR)**: The operator replaces the Priority constant parameter of a LocationRequest builder instance with a custom String value or an existing Priority constant. Similar to the previous operator, it is only applied to Location.

5.6.2.2 Null Operators

This category encompasses those mutation operators that replace the current instantiation of an object with a *null* value. Following, a description of the operators is presented as well as an example of applying two null operators (see Figure 42).

```java
// Example of DefaultDeviceBuilderInstanceDeclaration
private void example_DDBID() {
    // ORIGINAL CODE
    LocationRequest locationRequest = new LocationRequest
    .Builder(LocationRequest.PRIORITY_HIGH_ACCURACY)
    .setIntervalMillis(5000)  // 5 seconds
    .build();

    // MUTANT
    LocationRequest locationRequest = new LocationRequest();
}

// Example of RandomActionIntentDeviceDefinition
private void example_RAIDD() {
    ...
    // ORIGINAL CODE
    Intent enableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);

    // MUTANT
    Intent enableIntent = new Intent(BluetoothAdapter.ACTION_DISCOVERABLE);
}

// Example of ReplaceCompatibleTypeDeviceGetMethods
private void example_RCTDGM() {
    ...
    // ORIGINAL CODE
    double latitude = location.getLatitude();

    // MUTANT
    double latitude = location.getLongitude();
}
```

Figure 41 – Example of mutants generated by three different Replacement operators.

- **BuggyDeviceCallback (BDC)**: The operator changes the current instantiation of a Bluetooth or Location callback to a null value. That is, the callback has no response to an event or user interaction.

- **BuggyDeviceListener (BDL)**: The operator changes the current instantiation of a Bluetooth or Location listener to a null value. Similar to the previous operator, the listener has no response to an event or user interaction.

- **NullDeviceInstanceDeclaration (NDID)**: The operator changes a current instance declaration of a Bluetooth or Location class to a null value. This operator can be associated with operators NullGpsLocation and NullBluetoothAdapter from Linares-Vásquez et al. (2017) and NullLocation and NullBluetoothAdapter from the study of Liu et al. (2020).

- **NullReferenceDeviceMethods (NRDM)**: The operator changes the reference parameters of a method with a null value. This operator tackles method overloading.

```
// Example of BuggyDeviceListener
private void example_BDL() {
    ...
    // ORIGINAL CODE
    LocationListener locationListener = new LocationListener() {
        @Override
        public void onLocationChanged(Location location) {
            ...
        }
    };

    // MUTANT
    LocationListener locationListener = null;
}

// Example of NullReferenceDeviceMethods
private void example_NRDM() {
    ...
    // ORIGINAL CODE
    locationManager.requestLocationUpdates(
         LocationManager.GPS_PROVIDER,
         5000,
         10,
         locationListener
    );

    // MUTANT
    locationManager.requestLocationUpdates(
         LocationManager.GPS_PROVIDER,
         5000,
         10,
         null
    );
}
```

Figure 42 – Example of mutants generated by two different Null operators.

5.6.2.3 Switch Operators

This category encompasses two operators: SCDM and SCPDM. This type of operator switches the boolean value of a method parameter or changes the boolean return value of a method call. A description of each operator is presented below. Figure 43 shows an example of applying SCDM and SCPDM.

- ❑ **SwitchConditionalDeviceMethod (SCDM)**: The operator switches the boolean return value of a Bluetooth or Location method to “true” and “false”. That is, the operator creates one mutant with a “true” value and another with a “false” value. Additionally, it can be associated with operators UAB from Jabbarvand e Malek (2017) and BluetoothAdapterAlwaysEnabled from Linares-Vásquez et al. (2017).

- ❏ **SwitchConditionalParameterDeviceMethod (SCPDM)**: The operator switches the boolean parameter of a Bluetooth or Location method. Similar to the previous operator, it switches the value "true" to "false" and "false" to "true".

```
// Example of SwitchConditionalDeviceMethod
private void example_SCDM() {
    ...
    // ORIGINAL CODE
    boolean btEnabled = btAdapter.isEnabled();

    // MUTANT
    boolean btEnabled = true;
}

// Example of SwitchConditionalParameterDeviceMethod
private void example_SCPDM() {
    ...
    // ORIGINAL CODE
    beaconManager.setEnableScheduledJobs(false);

    // MUTANT
    beaconManager.setEnableScheduledJobs(true);
}
```

Figure 43 – Example of mutants generated by two different Switch operators.

5.6.2.4 Trap Operators

This operator inserts a trap method to reveal the reachability of a code in the application (AGRAWAL et al., 1989). It is specifically inserted into the callback or listener interface method. Whenever the trap method is executed, the mutant is killed and hence, the callback or listener method is reachable. A brief description of the operators is presented, and Figure 44 exemplifies their application.

- ❏ **TrapDeviceCallback (TDC)**: The operator inserts a trap method inside a callback.
- ❏ **TrapDeviceListener (TDL)**: The operator inserts a trap method inside a listener method.

5.6.2.5 Deletion Operator

This type of mutation operator deletes a statement declaration of the application. Below is a description of DDM and an example of its application (see Table 45).

❑ **DeletionDeviceMethods (DDM)**: The operator deletes a statement of methods that close, stop, and/or deallocate an existing process or service. It may cause collateral effects such as increasing the energy consumption of an application. This operator is associated with operators RLU and RBD from Jabbarvand e Malek (2017).

```
// Example of TrapDeviceCallback
private void example_TDC() {
    ...
    // ORIGINAL CODE
    private ScanCallback bleScanCallback = new ScanCallback() {
        @Override
        public void onScanResult(int callbackType, ScanResult result) {
            super.onScanResult(callbackType, result);
            ...
        }
    };

    // MUTANT
    private ScanCallback bleScanCallback = new ScanCallback() {
        @Override
        public void onScanResult(int callbackType, ScanResult result) {
            super.onScanResult(callbackType, result);
            TRAP_ON_CALLBACK();
            ...
        }
    };
}

// Example of TrapDeviceListener
public void example_TDL() {
    ...
    // ORIGINAL CODE
    LocationListener locationListener = new LocationListener() {
        @Override
        public void onLocationChanged(Location location) {
            ...
        }
    }

    // MUTANT
    LocationListener locationListener = new LocationListener() {
        @Override
        public void onLocationChanged(Location location) {
            ...
            TRAP_ON_LISTENER();
        }
    }
}
```

Figure 44 – Example of mutants generated by two different Trap operators.

```
// Example of DeletionDeviceMethods

// ORIGINAL CODE
private void example_DDM() {
    ...
    fusedLocationProviderClient.flushLocations();
    ...
}

// MUTANT
private void example_DDM() {
    ...

    ...
}
```

Figure 45 – Example of a mutant generated by DDM.

5.6.2.6 Shift Operators

This type of mutation operator moves the declaration of a method to another place in the source code. Following is a description of SDMC and an example of its application (see Figure 46).

- ❏ **ShiftDeviceMethodCall (SDMC)**: The operator shifts a Bluetooth or Location method call to another part of the source code. It can be shifted into the same method or in a different method from the same class.

5.6.3 Discussion

In the previous section, all 16 mutation operators were properly described. For space reasons, we did not present an example for all mutation operators. See Section "Data Availability" to access the URL link to the repository containing all supplementary data collected in this work.

The 16 mutation operators were grouped into six categories (Replacement, Null, Switch, Trap, Deletion, and Shift) to mimic the possible faults that may occur during development. Note that the proposed operators may not cover all Bluetooth, AltBeacon, and Location functionalities. To minimize this possible threat, we focused on the main functionalities of each component. Additionally, the mutation operators were designed to be generic and hence, it could be extended for other functionalities and mobile device resources.

Additionally, we cross-checked the proposed operators against those previously defined in the academic literature. As observed, six of these operators (DVO, RLPDR, DDM, RCTDGM, SCDM, and NDID) have subsumed the existing ones, whereas ten of them are new contributions to the present work.

```
// Example of ShiftDeviceMethodCall

// ORIGINAL CODE
private void example_SDMC() {
    @Override
    public void onCreate(Bundle savedInstanceState) {
       ...
    }

    @Override
    public void onPause() {
       super.onPause();
       bluetoothServerSocket.close();
    }
}

// MUTANT
private void example_SDMC() {
    @Override
    public void onCreate(Bundle savedInstanceState) {
       ...
       bluetoothServerSocket.close();
    }

    @Override
    public void onPause() {
       super.onPause();
    }
}
```

Figure 46 – Example of two mutants generated by SDMC.

Finally, when observing the characteristics of the proposed mutation operators, we found that parametrization is essential for some operators. For example, the mutation operator RandomActionIntentDeviceDefinition (RAIDD) requires a valid action intent from the same class. That is, if the operator mutates an action intent from "BluetoothAdapter", it is expected that the mutation operator replaces the intent with another one from the "BluetoothAdapter" class.

Note that parametrization makes the approach more flexible because it provides to the developer the possibility of creating a set of possible mutation operators to be applied. That is, the developer controls the intent, methods, or values to be mutated.

5.7 Empirical Cost Evaluation

This section provides an empirical cost evaluation of applying the mutation operators by assessing the number of mutants generated per application and the number of mutants generated per operator. For this evaluation, two subject apps were selected. Moreover, these apps were adopted in other studies (JABBARVAND; MALEK, 2017; PAN et al., 2020):

1. **a2dpvolume**[4]: This app automatically adjusts the media volume on connect and resets on disconnect.

2. **runnerup**[5]: This app tracks sports activities using GPS from Android devices.

Each app was carefully evaluated, and the mutants were manually generated whenever a possible mutation point was found. For this purpose, the mutation point was replaced with a possible mutant, and the code was recompiled to check the validity of the mutation. Note that all mutants compiled successfully.

In this analysis, only a single mutant was generated for the mutation point that allows applying the same mutation operators multiple times. For instance, the operator RAIDD may generate a large number of mutants considering a mutation point, but only a single mutant is considered in the analysis. Therefore, this analysis provides the lower boundary mutants generated per app as presented in Table 40. From now on, we refer to the a2dpvolume application as APP_1 and to the runnerup application as APP_2.

Table 40 – Number of mutants generated per app.

App Id	Name	# Classes	# Mutated Classes	# Mutants
APP_1	a2dpvolume	22	7	86
APP_2	runnerup	150	12	51

As can be observed, the APP_1 is a smaller application but generates 35 more mutants than APP_2. One plausible explanation is that the APP_2 has fewer classes related to Location and Bluetooth that can be mutated compared to APP_1. Additionally, APP_1 generates mutants for Bluetooth and Location, whereas APP_2 generates only for Location. Moreover, none of the applications generated mutants for the AltBeacon API, which may reflect the lack of mutant generation for some mutation operators.

Next, Figure 47 presents the number of generated mutants per operator.

The results show that the operator RCTDGM generates the largest number of mutants for both apps. In the case of APP_1, 41 mutants were generated, representing a total of 47% of mutants. APP_2 generated 15 mutants, that is, 29% of all mutants. It is worth noting that the mutation operator was applied once per mutation location. Therefore, the number of mutants generated would increase if the operator replaces the call of the get methods from an object with all possible permutations.

The second operator that generated the most mutants was NDID. For APP_1, the operator generated 20 mutants (i.e., 23%) whereas APP_2 generated 14 mutants (i.e., 27%). Additionally, five mutation operators generated mutants for a single application. The operators BDL and TDL only mutated APP_1 while the operators BDC, TDC, and DVO mutated APP_2.

[4] https://github.com/jroal/a2dpvolume
[5] https://github.com/jonasoreland/runnerup

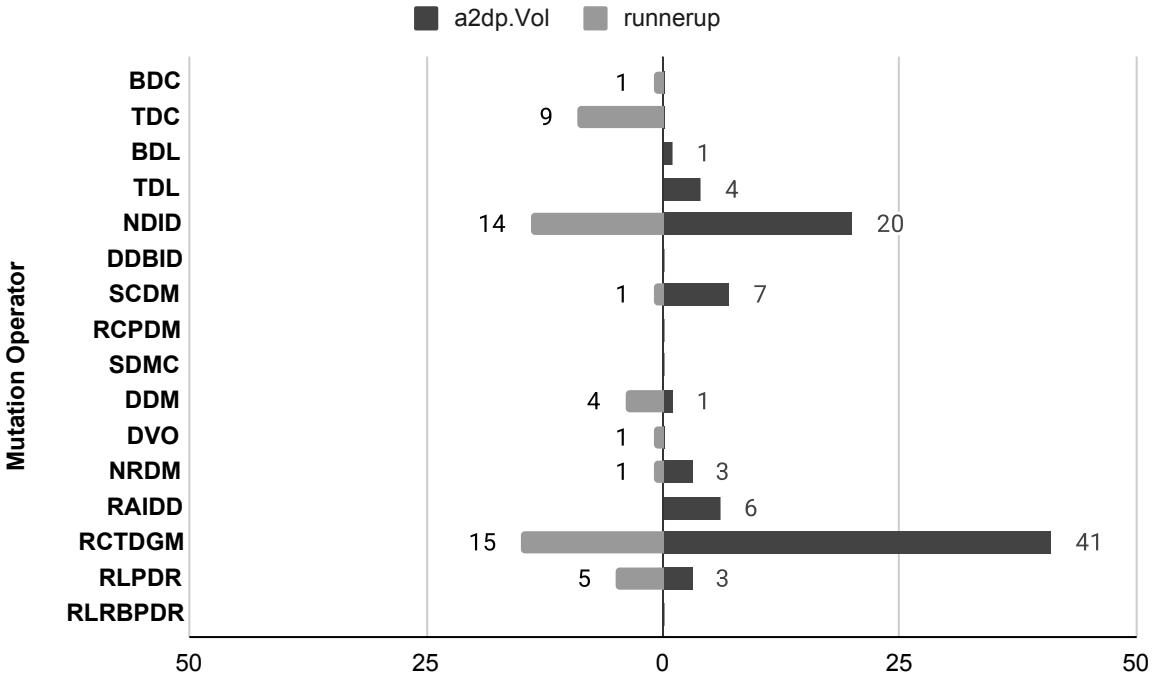


Figure 47 – Number of mutants generated per operator.

On the other hand, four operators did not generate any mutants for the two applications: DDBID, RCPDM, SDMC, and RLRBPDR. Note that this does not imply that these are useless mutation operators. For instance, the RCPDM operator is useful for applications that rely on the AltBeacon library due to the API structure. Similar to RLRBPDR, which requires the instantiation of a *LocationRequest.Builder* method to generate the mutant. In this case, it was possible to observe that the subject apps did not implement those functionalities that the operators mutate and hence, the statistics may change if a broader set of subject apps is considered.

Observe that this analysis only focuses on the viability of the mutation operators, i.e., if they are capable of generating mutants for Bluetooth and Location applications. Therefore, it is expected that in the future, an in-depth evaluation of the proposed mutation operators regarding their usefulness, i.e., subsumption, equivalence, and triviality (JUST; KURTZ; AMMANN, 2017).

5.8 Related Work

5.8.1 Android Mutation Operators

Silva et al. (2022) presented a systematic mapping study assessing the studies related to mutation testing for mobile applications. As a result, the study found that a total of 138 mutation operators were proposed across 16 primary studies, as the majority focus on configuration and GUI aspects.

On the other hand, few studies focus on other mobile device components such as connectivity, location, and sensors. That is, only four studies proposed at least one mutation operator for these components. Jabbarvand e Malek (2017) investigated mutation operators aiming at energy consumption. That is, the faults are modeled to verify whether they may result in a difference in energy consumption between the mutant and the original AUT. In total, 50 mutation operators were designed, of which eight related to Location and four related to Bluetooth.

Linares-Vásquez et al. (2017) proposed an in-depth study that defines a taxonomy of the main Android faults. Then, 38 mutation operators were devised based on the taxonomy. One operator related to the Location injects a null GPS location. Additionally, two mutation operators are associated with Bluetooth: one makes the "BluetoothAdapter" instance always enabled, whereas the other instantiates "BluetoothAdapter" with a null value.

Deng, Offutt e Samudio (2017) designed 17 mutation operators categorized as: Event-based, Component Lifecycle, XML-related, Common Faults, Context-aware, Energy-related, and Network-related. From this set, only one relates to Location, which mutates attributes such as latitude, longitude, altitude, and speed.

Finally, Liu et al. (2020) proposed 32 mutation operators divided in Android-specific operators and Java-specific operators. Only one Bluetooth mutation operator was proposed that instantiates "BluetoothAdapter" with null, similar to the operator proposed by Linares-Vásquez et al. (2017).

5.8.2 HAZOP for Mutation Operators Design

As discussed, there is still no standardized methodology for mutation operator design. However, some initiatives try to use existing systematic approaches aiming at minimizing any possible bias.

The Hazard and Operability approach, or HAZOP, was applied in some studies to facilitate the mutation operator design. Kim, Clark e Mcdermid (2000b) uses HAZOP to devise mutation operators for Java programs.

Araujo et al. (2011) applied HAZOP to design mutation operators for Dynamic System Models totaling 12 operators. Savarimuthu e Winikoff (2013) applied HAZOP for the GOAL language, devising 21 mutation operators. Zhang et al. (2016) proposed 191 mutation operators for Restricted Use Case Modelling (or RUCM).

5.9 Conclusion

This paper provided an initial investigation of the design of Android mutation operators for specific components, i.e., Bluetooth and Location. This study considered the Android native API and the third-party library AltBeacon. The rationale is that the present study is being conducted in an academia-industry partnership in which the company develops applications that heavily rely on these components.

Overall, this work resulted in 16 different Bluetooth and Location mutation operators by using a systematic approach named HAZOP. The operators were categorized based on their types: Replacement, Null, Switch, Trap, Deletion, and Shift. Additionally, we validated the mutation operators by manually generating the mutants, considering two Android applications showing the validity and feasibility of the operators, i.e., their capabilities of generating real faults related to these two components.

By providing a generic set of mutant operators, we advocate that the extension of the mutation operators for other mobile device components, such as Wi-Fi, NFC, and sensors, would be easily carried out. All these components have an API structure similar to Bluetooth and Location, thus, many operators can be reused for these device resources. For instance, they rely on callback or listener functions to handle events or actions, thus, it is possible to extend the operators BDC, BDL, TDC, and TDL to handle these device components. Therefore, we intend to extend the set of mutation operators once they are validated in a controlled experimental environment and an industrial scenario.

Regarding automation aspects, we intend to implement the mutation operators for automatic generation and execution of the mutants. As presented by Silva et al. (2022), there are still few tools that support mutation testing for mobile applications. In fact, only three of them support all stages of mutation testing: generation, execution, and analysis. Recently, Vincenzi et al. (2025) proposed METFORD, a mutation testing framework that implements a set of mutation operators considering the Mutation Schemata (UNTCH; OFFUTT; HARROLD, 1993; POLO-USAOLA; RODRíGUEZ-TRUJILLO, 2021) and traditional approach.

In an initial analysis, METFORD appears to address the implementation requirements of the proposed operators, as it is an open-source tool that supports the parametrization of mutation operators. That is, it allows the tester to set mutation operators to be applied and their parameters. For instance, one can set the scalar values to be incremented and decremented from the DVO operator. This provides a more controllable environment for

the tester to inject all the desired faults. Additionally, this approach also benefits the reuse of the proposed mutation operators for other device components without enlarging the existing set.

We intend to validate the mutation operators through experimentation using open-source Android applications and conduct case studies in an industrial scenario. The main difficulty is finding Bluetooth, Location, or AltBeacon apps with implemented tests. Pecorelli et al. (2021) conducted an empirical study and found that few Android applications (approximately 40% of the evaluated apps) contain at least one test suite. Moreover, Vincenzi et al. (2025) showed the poor quality of existing test suites of Android applications, given the defect models proposed in their work. Therefore, we will explore strategies to establish a viable benchmark for validating the mutation operators.

ARTIFACT AVAILABILITY

The following repository contains all the supplementary data collected from this work: <https://github.com/phkuroishi/paper-sbes-device-mutation-operator>.

Chapter 6

Automated Android Instrumented Unit Test Generation using Large Language Models for Bluetooth and Location Components: An Experimental Study

6.1 Overview

This chapter presents the last paper that ends the PhD timeline. Unlike the others, this article is still in the submission process. We aim to submit this paper to a Special Issue by November 15, 2025. Since this paper is still under the submission process, there may be some differences between the manuscript presented in the thesis and the submitted version. However, we pinpoint that it does not affect the results of this work.

In the previous paper, we devised a set of 16 mutation operators that consider the Bluetooth (i.e., native and AltBeacon) and Location components of the Android API. Now, the present paper also contributes to optimizing the testing process by showing how we can use such a fault model to evaluate and generate test cases for Android applications, aiming to detect faults related to Bluetooth and Location components.

In this paper, we explored the capabilities of existing large language models (LLMs) to generate instrumented unit test cases for the Bluetooth and Location APIs of the Android ecosystem in an automated manner. Additionally, the test cases were evaluated in three dimensions: using our fault model, code coverage, and the chrF similarity metric.

Below is the information regarding the paper:

- **Title:** Automated Android Instrumented Unit Test Generation using Large Language Models for Bluetooth and Location Components: An Experimental Study.
- **Authors:** Pedro Henrique Kuroishi (UFSCar), Ana Cristina Ramada Paiva (Universidade do Porto), José Carlos Maldonado (ICMC-USP), and Auri Marcelo Rizzo Vincenzi (Universidade do Porto and UFSCar).
- **Local:** Empirical Software Engineering – Advancing Software Engineering with Large Language Models – Special Issue.
- **Year:** 2025
- **Status:** To be submitted – deadline: November 15, 2025.

6.2 Abstract

Purpose: The gain of popularity of GenAI changed the paradigm of how Software Engineering tasks are carried out. In software testing, many studies have proposed methods to optimize test case generation using large language models (LLMs). In the Android ecosystem, most research focuses on GUI test case generation, whereas little attention is paid to other Android hardware components, such as connectivity, location, and sensors. Thus, this paper aims to provide an initial investigation of Android instrumented unit test case generation for Bluetooth and Location.
Methods: To carry out this study, we conducted an experimental study to assess the capabilities of existing LLMs in generating instrumented unit test cases for the components mentioned above. In this case, two strategies were adopted: (i) starting from scratch and (ii) mutation-guided test case generation. Additionally, the tests were evaluated based on three metrics: (i) code coverage, (ii) mutation testing, and (iii) the chrF similarity metric.
Results: The results show that Claude had the best performance regarding code coverage and mutation score, whereas ChatGPT had an interesting code coverage performance but poorly performed in mutation testing. DeepSeek and Qwen have the potential to complement existing test cases. Moreover, chrF shows little similarity across the test sets generated by each LLM. Finally, the mutation-guided test case generation may be an interesting approach to enhance the existing test set by focusing on specific defects.

Conclusion: This paper provides an initial investigation towards generating unit test cases for specific Android device components (i.e., Bluetooth and Location). Despite the promising results, there is still room for improvement. For instance, explore GUI testing with the components mentioned before, expand the study to other Android components, and explore different metrics, such as test smells and sustainability-related metrics.

6.3 Introduction

Generative Artificial Intelligence, or simply GenAI, is changing the paradigm of how Software Engineering tasks are carried out. From software specification to maintainability, the use of large language models (LLMs) is increasingly widespread in academic research tackling these topics (ZHENG et al., 2024). Moreover, some reports estimate that 71% of organizations have already integrated GenAI into their operational environment and that 34% have adopted it for quality assurance activities (CAPGEMINI, 2025).

Software testing is a crucial part of the software development life cycle. Despite its importance, testing is considered a very costly task (MYERS; SANDLER; BADGETT, 2011; LUO, 2001), especially designing test cases, which is often considered a laborious and error-prone activity (POTUZAK; LIPKA, 2023; WANG et al., 2021). In this sense, automation is essential for optimizing this process. Over the years, several studies have investigated different strategies for automatic test case generation (ZENG et al., 2016; RAMLER; KLAMMER; BUCHGEHER, 2018; WANG et al., 2021; CHOUDHARY; GORLA; ORSO, 2015). And even with acknowledged advances in the field, there is still difficulty in producing correct syntactic and semantic test cases for all types of applications (WANG et al., 2025).

The rise of LLMs appears to offer an alternative for enhancing automatic test case generation. Recently, many studies have begun investigating the capabilities of existing LLMs to generate unit test cases (YI et al., 2023; SIDDIQ et al., 2024; CHEN et al., 2025b; NAN et al., 2025). In the Android ecosystem, most studies focus on validating the GUI aspects of the application (LIU et al., 2023b; GARCÍA et al., 2024; LIU et al., 2024; YOON et al., 2025). Somehow, this is expected given the event-based characteristics of Android applications (AMALFITANO et al., 2013; AMALFITANO et al., 2015). On the other hand, we found a few community efforts in investigating test case generation for different hardware components, such as connectivity, location, and sensors.

Therefore, the paper aims to contribute an initial investigation of the capabilities of existing LLMs to generate Android instrumented unit tests for two Android components:

Bluetooth and Location. In the context of Android, instrumented tests are designed to run on an emulator or real device[1]. Given the limitations of the emulator[2], an experimental study was executed using a real Android device.

The tests were generated considering four LLMs: Claude, ChatGPT, DeepSeek, and Qwen. Moreover, the experimental study was executed considering two strategies. In the first strategy, we assumed the applications lacked tests. In fact, the study by Pecorelli et al. (2021) shows that most open-source Android applications lack a single test set. Therefore, it helps us justify adopting a test-generation approach from scratch. In the second strategy, we adopted a mutation-guided test case generation, somewhat similar to what is presented in Harman et al. (2025). In this case, the plan aims to generate tests focusing on undetected faults.

We leverage three metrics to evaluate the test cases: code coverage, mutation score, and chrF similarity metric (POPOVIĆ, 2015). The first two are well-known, commonly used metrics for assessing quality. The latter metric evaluates the degree of similarity among the generated test sets:

In summary, this paper provides the following contributions:

- ❏ Investigate the capabilities of Claude, ChatGPT, DeepSeek, and Qwen in generating Android instrumented unit tests for Bluetooth and Location components.
- ❏ Compare the generated test sets based on code coverage, mutation score, and chrF similarity metric.
- ❏ Evaluate the mutation-guided test case generation approach to enhance the generated test set.

The rest of the paper is structured as follows: Section 6.4 presents the experimental goals and design. Section 6.5 describes the results of the experimental study. Section 6.6 presents the second phase of the experimental research. Section 6.7 demonstrates a qualitative analysis of the implemented mutation operators. Section 6.8 discusses and provides insights into the results. Section 6.9 presents the studies related to this work. Section 6.10 discusses the threats to the validity, whereas Section 6.11 concludes this work.

6.4 Experimental Study Design

This section describes the steps to set up the experimental environment for test case generation.

[1] https://developer.android.com/training/testing/instrumented-tests
[2] https://developer.android.com/studio/run/advanced-emulator-usage

6.4.1 Study Goal

The main goal of this study is to assess the capability of existing large language models in generating Android instrumented unit tests for two device components: Bluetooth and Location. Therefore, the following research question (RQ1) is defined:

RQ1: What is the capability of large language models in generating Android instrumented unit tests for Bluetooth and Location components?

6.4.2 Study Design

6.4.2.1 Experimental Study Workflow

Figure 48 illustrates the experimental study workflow.

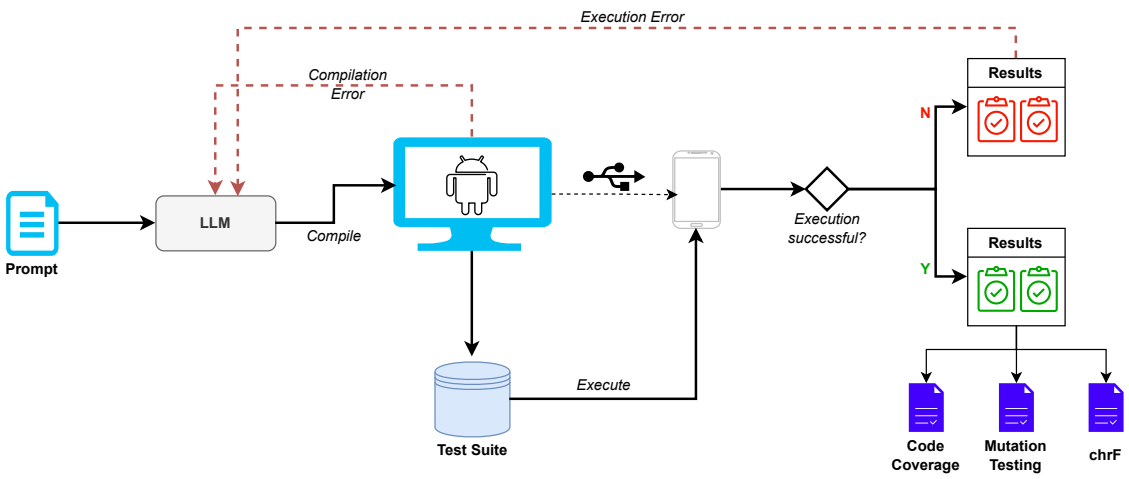


Figure 48 – Experimental study workflow.

The process starts by running the defined prompt on each LLM selected for the experimental study. Then, the tests were compiled, and whenever a compilation error occurred, a new interaction with the LLM was initiated to resolve it. Next, the tests were executed using a real mobile device. If the tests failed to run, a new interaction with the LLM was conducted to fix the error. For this stage, we set a maximum of three interactions with the LLMs. We observed that after the third interaction, LLM began to hallucinate. If no compilable or executable test set was provided, we marked the failed tests with the @Ignore tag for statistical purposes. Finally, we computed the following metrics: code coverage, mutation score, and chrF similarity metric. We detailed each metric in further sections.

Moreover, we ran the test sets five times to minimize the effects of flakiness, and the final results were the average values for each metric. In case of flakiness, we tagged the test with *@Ignore("FLAKY_TEST")*, and maintained it in the test set for statistical purposes.

6.4.2.2 Subject Apps Selection

The following requirements were considered during the subject apps selection process:

- ❏ The app must be a native Android application.
- ❏ The app must be related to the Bluetooth and/or Location category. That is, the app must implement or use these two API's.
- ❏ The app must be written primarily in Java, especially for components that use the Bluetooth and Location APIs.
- ❏ The application's source code must be available.
- ❏ The application must have been updated within the last 5 years.
- ❏ The app must compile with Java compiler version 11.
- ❏ The app must have at least 100 stars (>= 100). In this case, the number of forks is optional.

We considered both GitHub[3] and F-Droid[4] repositories during the app selection phase. In the first repository, we performed the search using the following keywords: "android bluetooth", "android location", "android gps", "android ble", "bluetooth", "ble", "location" and "gps". Moreover, we set the filter *Languages* to "Java" and *Number of Stars* to ">=100". In the latter repository, we manually searched apps considering the following categories: Connectivity, Navigation, and Sports & Health.

A candidate application was cloned into a local machine and compiled using Java 11. The rationale for using this version is that we evaluated the quality of the generated tests using the mutation testing framework METFORD, which requires the application to be built using Java 11. Moreover, there were cases where the application version was downgraded to be compatible with Java 11. At the end of the process, 10 Android apps were selected for this study. Table 41 shows the information about the applications.

[3] https://github.com
[4] https://f-droid.org

Table 41 – Subject apps used in the experimental study.

App	Acronym	Last Update	Version	# Stars	# Fork
a2dpvolume[5]	A2DP	2019*	2.13.0.4	102	33
AndrOBD[6]	AOBD	2025	v2.6.15	1700	352
Android-Scanner-Compat-Library[7]	ASCL	2021	1.6.0	784	161
Authorizer[8]	AUTH	2023	v0.5.0	548	60
codec2_talkie[9]	C2T	2025	v1.81	258	40
GadgetBridge[10]	GB	2025	0.80.0	1300	477
gpslogger[11]	GPSL	2025	v131	2200	623
runnerup[12]	RU	2024	v2.8.0.0	835	283
ulogger-android[13]	ULOG	2025	v3.12	332	56
WarpShare[14]	WS	2023	v2.0.4	641	39

Note that a2dpvolume did not meet the criteria that require an app to be updated in the last five years. Despite this issue, we decided to keep it in the final set because this application has been widely adopted in other studies. Additionally, the app was implemented using the legacy Android Support library, which AndroidX later replaced. Therefore, we forked the application and updated it to AndroidX, preserving all the app's original functionality.

Furthermore, we selected a subset of classes for the experimental study. That is, only those classes that directly or indirectly implement the Bluetooth and Location components of the Android API were considered for the study. Table 42 lists the classes of each application, totaling 58 classes. From now on, we refer to each subject app by its acronym.

6.4.2.3 LLM Selection

In the present study, we considered four large language models widely adopted in other studies: Claude, DeepSeek, Qwen, and GPT-5. Moreover, we leveraged OpenRouter[15] to centralize execution and facilitate data collection for the following LLMs: DeepSeek, Qwen, and GPT-5. For Claude Sonnet 4, we used its free quota. We requested tests until it reached its day token limit, then shifted to OpenRouter to continue data collection. Additionally, all tests were requested via the graphical interfaces of OpenRouter and Claude, as we found it easier to interact with the LLM without losing context. Table 43 describes the LLMs used in the experimental study.

[5] https://github.com/jroal/a2dpvolume
[6] https://github.com/fr3ts0n/AndrOBD
[7] https://github.com/NordicSemiconductor/Android-Scanner-Compat-Library
[8] https://github.com/tejado/Authorizer
[9] https://github.com/sh123/codec2_talkie
[10] https://codeberg.org/Freeyourgadget/Gadgetbridge
[11] https://github.com/mendhak/gpslogger
[12] https://github.com/jonasoreland/runnerup
[13] https://github.com/bfabiszewski/ulogger-android
[14] https://github.com/moseoridev/WarpShare
[15] https://openrouter.ai/

Table 42 – Elected classes per app.

App	Class
A2DP	Connector
AOBD	MainActivity
	BtCommService
	BtDeviceListActivity
ASCL	ScanFilter
	PendingIntentReceiver
	BluetoothLeScannerImplOreo
	BluetoothLeScannerImplLollipop
	BluetoothLeScannerImplJB
	BluetoothLeScannerImplMarshmallow
AUTH	BluetoothDeviceListing
	HidDeviceController
	HidDeviceProfile
	BluetoothUtils
	HidDeviceApp
	BluetoothForegroundService
C2T	BleConnectActivity
	Position
	Periodic
	BleGattWrapper
	Smart
	BluetoothConnectActivity
GB	BluetoothStateChangeReceiver
	GBDeviceCandidate
	BluetoothPairingRequestReceiver
	PhoneNetworkLocationProvider
	GBLocationListener
	DeviceManager
	BondingUtil
	BluetoothConnectReceiver
	PhoneGpsLocationProvider
	BtBRQueue
	DeviceHelper
GPSL	Gpx10FileLogger
	GeneralLocationListener
	CustomUrlManager
	Kml22FileLogger
	GpsLoggingService
	Gpx11FileLogger
	GeoJSONLogger
	CSVFileLogger
	Locations
	Maths
	Session
	GpxReader
	GeoJSONWriterPoints
RU	TCX
	TrackerGPS
	GpsStatus
	ActivityCleaner
	PathSimplifier
ULOG	LoggerService
	LocationFormatter
	LoggerTask
	LocationHelper
WS	ReceiverServiceClaude
	ConfigManagerClaude
	AirDropBleController

Table 43 – Description of the LLMs.

Type	Name	Developed by
Free	Deepseek V3 0324	High-Flyer
	Qwen3 Coder	Alibaba Cloud
Subscription	Claude Sonnet 4	Anthropic
	GPT-5	OpenAI

OpenRouter offers free versions of the DeepSeek and Qwen models. The main differences between the free and paid versions are the number of providers, which is constrained compared to the paid version, and the limited number of exchanged tokens and requests per minute/day. Moreover, the free version may use previous prompts and interactions to improve new models. In theory, this constraint does not affect the experimental results since it focuses on new models. We pivoted to the paid version of DeepSeek and Qwen whenever we reached their limits.

6.4.2.4 Prompt Design

The process started with prompt designing. In the first attempt, we defined the following information for prompting composition:

- ❏ *Role*: Senior Android developer with broad experience in code coverage testing.
- ❏ *Code*: The class under test.
- ❏ *Project Structure*: The project structure considering the packages and classes
- ❏ *Constraints*: Android instrumented tests, JUnit 4, avoid additional comments and explanations, avoid using mocks, test class name, and test package.

To validate the prompt, we randomly selected a class of one of the apps used in the experimental study and Claude Sonnet 4 as the LLM. The prompt was structured as a text file and presented to the LLM via its interactive interface. During this step, we successfully obtained an executable test set.

Next, we decided to remove the project structure to minimize the number of tokens. Since the results were similar to the initial prompt, the project structure was removed from the prompt. Figure 49 exemplifies the prompt used in the experimental study.

```
Your role is a senior Android developer with broad experience in code
coverage testing. I need an Android instrumented test suite in JUnit 4. Below
is the class under test -- <FULLY_QUALIFIED_CLASS_NAME>:
----------------------------------
<CLASS_UNDER_TEST>
----------------------------------
Generate the Android instrumented unit test in JUnit 4 format without
comments and additional explanations with tests instantiating the
<CLASS_UNDER_TEST> class without using mocks. The testing class must call
<TEST_CLASS_NAME> and the test package is: <TEST_PACKAGE>.
```

Figure 49 – Prompt defined.

To facilitate further analysis, we created a test package for each LLM and standardized the test class name as follows: $CLASS\_UNDER\_TEST + LLM + Test$. For instance, *LocationFormatterClaudeTest*.

Additionally, we used a different prompt whenever the tests failed to compile or execute, and we provided the error output. Observe that, when using the interactive version of the LLM, we do not need to provide the complete source code of the generated test set, as it is available to the LLM in the context window. Figure 50 shows this prompt snippet:

```
The tests failed to <COMPILE/EXECUTE> due to:
<ERROR_CODE>
```

Figure 50 – Snippet of the prompt used for LLM interaction.

6.4.3 Code Coverage

Code coverage was computed using the JaCoCo tool. In this paper, we collected this metric using three different strategies:

- *Per LLM*: Evaluate the code coverage of each LLM individually.
- *Pairwise*: Evaluate the code coverage by combining a pair of LLMs' generated test sets. In this case, we consider all possible combinations of LLMs' generated test sets, that is, *(Claude, DeepSeek)*, *(Claude, ChatGPT)*, *(Claude, Qwen)*, *(ChatGPT, DeepSeek)*, *(ChatGPT, Qwen)*, and *(DeepSeek, Qwen)*.
- *Incrementally*: Evaluates if combining the generated test sets of different LLMs increases code coverage. In this case, we considered the highest coverage to the lowest coverage, and vice versa.

6.4.4 Mutation Testing

This section presents the quality assessment of the generated tests through mutation testing. For this purpose, we performed the following steps:

1. Implement the mutation operators.
2. Generate and execute the mutants.
3. Compute mutation score.

Additionally, a similar code coverage analysis (i.e., per LLM, pairwise, and incrementally) was carried out for mutation testing.

6.4.4.1 Mutation Operators Implementation

Kuroishi et al. (2025) proposed a set of 16 mutation operators for Android Bluetooth, AltBeacon, and Location API's. These operators were classified into six different groups: Replacement, Null, Switch, Trap, Deletion, and Shift. For the present paper, we randomly selected and implemented 10 out of 16 mutation operators. Table 44 presents the list of the proposed mutation operators, and those highlighted in gray are the ones we implemented.

Table 44 – Implemented mutation operators (extracted from Kuroishi et al. (2025)).

Category	Operator	Name
Replacement	**DDBID**	**DefaultDeviceBuilderInstanceDeclaration**
	DVO	DeviceVariablesOperator
	RAIDD	**RandomActionIntentDeviceDefinition**
	RCTDGM	ReplaceCompatibleTypeDeviceGetMethods
	RLPDR	**RandomLocationProviderDeviceReplacement**
	RLRBPDR	**RandomLocationRequestBuilderPriorityDeviceReplacement**
Null	**BDC**	**BuggyDeviceCallback**
	BDL	**BuggyDeviceListener**
	NDID	**NullDeviceInstanceDeclaration**
	NRDM	NullReferenceDeviceMethods
Switch	SCDM	SwitchConditionalDeviceMethod
	SCPDM	SwitchConditionalParameterDeviceMethod
Trap	**TDC**	**TrapDeviceCallback**
	TDL	**TrapDeviceListener**
Deletion	**DDM**	**DeletionDeviceMethods**
Shift	SDMC	ShiftDeviceMethodCall

The mutation operators were implemented into the METFORD framework (VINCENZI et al., 2025). The tool applies mutation testing to the Java source code and adopts the Mutation Schemata approach (UNTCH; OFFUTT; HARROLD, 1993). In this case, all mutants are generated within the same code base and, consequently, stored in a single APK file.

Additionally, some mutation operators were designed to be parametrized for greater flexibility (KUROISHI et al., 2025). Therefore, we adopt the same strategy when implementing the mutation operators. Table 45 displays information about the operators and their possible parameters.

Table 45 – List of parameterized mutation operators highlighted in gray.

Mutation Operator	Parameterized	Parameter
BDC	N	
BDL	N	
DDBID	N	
DDM	Y	Method name to be mutated
NDID	Y	Method name to be mutated
RAIDD	Y	Random seed
RLPDR	Y	Random seed
RLRBPDR	Y	Random seed
TDC	N	
TDL	N	

6.4.4.2 Mutants Generation

As presented in Table 45, some operators are parameterized, i.e., they allow using the same operator with distinct parameter values. In METFORD (VINCENZI et al., 2025), one must define a configuration JSON file that specifies the mutation operators to apply and any optional parameters. Figure 51 shows a JSON snippet of the METFORD configuration file with the mutation operators used in the experiment.

As can be observed, the *operatorNameList* key contains a list of the mutation operators, whereas the *operatorArgumentList* key has a list of the parameters associated with each operator. Note that some mutation operators may be repeated due to their parametrization support.

The operators with empty brackets do not use parametrization. For *RandomActionIntentDeviceDefinition*, *RandomLocationProviderDeviceReplacement*, and *RandomLocationRequestRequestBuilderPriorityDeviceReplacement*, we randomly selected a number ranging from 1 to 10 as the seed. We defined the methods and classes to be mutated for *DeletionDeviceMethods* and *NullDeviceInstanceDeclaration*.

Moreover, the mutants were generated using the same set of mutation operators and parameter values for all ten Android apps. In this paper, we did not detail the implementation of the mutation operators. However, we pinpoint that we followed the same programming design as the existing METFORD operators. Any additional information about the technical aspects of the framework is broadly discussed in Vincenzi et al. (2025).

METFORD (VINCENZI et al., 2025) also allows selecting the project files to be mutated. In this case, we restrict the generation of mutants to those classes related to Bluetooth and Location.

We provided an online repository containing all supplementary data, including the source code of the mutation operators implemented in this paper.

6.4.4.3 chRF Similarity Metric

The character n-gram F-score metric can be used for machine translation output (POPOVIĆ, 2015). It may provide a metric of the similarity between a machine-generated output and a reference text. Recent studies adopted this metric to evaluate the quality of AI-generated test cases compared to a reference code (WYNN-WILLIAMS et al., 2025). In this paper, we also used this metric to assess the similarity among tests generated by the LLMs.

We standardized the test sets for each LLM to ensure a fair comparison. First, we considered only the successfully executed tests, i.e., those tagged with *@Test*. The tests tagged with *@Ignore* were promptly discarded due to compilation error, execution error, or flakiness. We also considered private methods associated with the executable tests and methods marked with: *@Before* and *@After*. Moreover, we discarded the class declaration

```
{
  "operatorNameList":[
      "BuggyDeviceCallback",
      "TrapDeviceCallback",
      "BuggyDeviceListener",
      "TrapDeviceListener",
      "DefaultDeviceBuilderInstanceDeclaration",
      "RandomActionIntentDeviceDefinition",
      "RandomLocationProviderDeviceReplacement",
      "RandomLocationRequestBuilderPriorityDeviceReplacement",
      "DeletionDeviceMethods",
      "DeletionDeviceMethods",
      "NullDeviceInstanceDeclaration",
      "NullDeviceInstanceDeclaration",
      "NullDeviceInstanceDeclaration",
      "NullDeviceInstanceDeclaration",
      "NullDeviceInstanceDeclaration",
      "NullDeviceInstanceDeclaration",
      "NullDeviceInstanceDeclaration"
    ],
    "operatorArgumentList":[
      [],
      [],
      [],
      [],
      [],
      [7],
      [7],
      [7],
      ["removeUpdates"],
      ["removeGpsStatusListener"],
      ["BluetoothAdapter"],
      ["BluetoothLeScanner"],
      ["BluetoothSocket"],
      ["BluetoothManager"],
      ["Location"],
      ["LocationManager"],
      ["LocationRequest"]
    ]
}
```


Figure 51 – JSON snippet of the configuration file for METFORD.

because we gave each test class a different name, which may skew the metric calculation. The test class was parsed into a plain-text file, with each test case on a separate line. Figure 52 exemplifies the parsed test set.

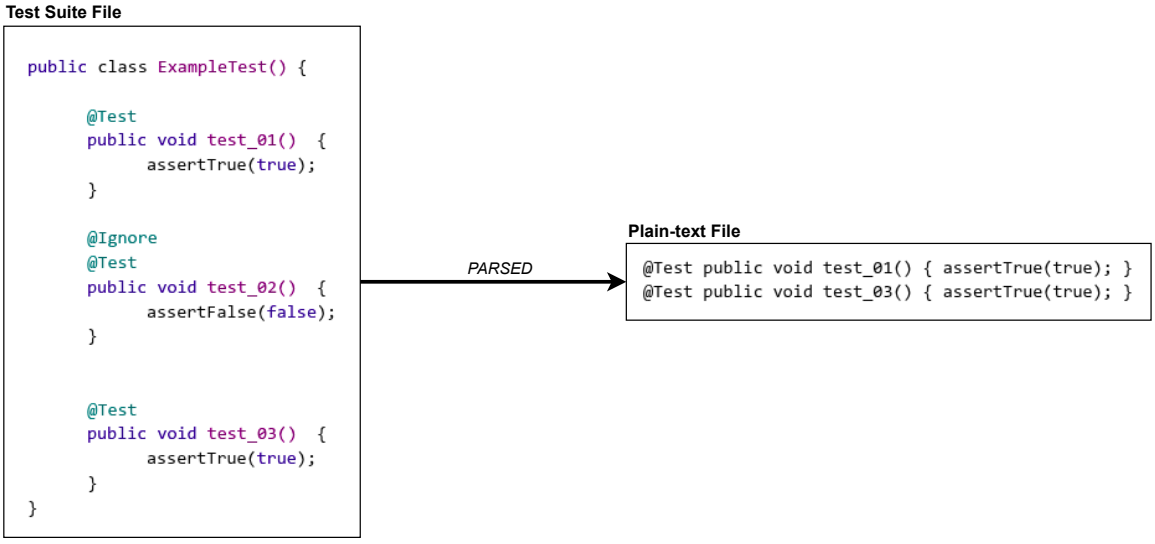


Figure 52 – Example of test set parsed file.

Note that different metrics for evaluating machine-code similarity, such as BLEU (PAPINENI et al., 2002) and CODEBLEU (REN et al., 2020), may also be adopted. However, Evtikhiev et al. (2023) evaluated that chrF is the most accurate metric compared to BLEU and CODEBLEU.

Finally, the following script was used to compute the chrF score using the default parameters and can be found at: <https://github.com/m-popovic/chrF>.

6.4.4.4 Environment Settings

The experimental study was performed on a computer with a Quad-Core Intel Core i5-9300H processor, 24 GB of RAM, and an NVIDIA GeForce GTX 1650 Mobile GPU. The tests ran on a real mobile device, a Motorola Moto G6 Play, with Android version 9. The device uses the factory configuration to prevent external applications from interfering with the process, and it is physically connected to the computer via the USB port.

Additionally, the projects were configured with Android Studio Meerkat 24.3.2 and compiled with Java 11, since METFORD (VINCENZI et al., 2025) is designed to run on this version.

6.5 Results

6.5.1 Test Case Generation using LLMs

This section presents information on the test cases generated by each subject app and LLM, along with the number of ignored tests per app. Initially, Table 46 presents

statistics of the test cases generated per app. Columns N_{TEST}, N_{IGN}, and N_{EXEC} present the total number of tests generated per app, the number of ignored tests, and the number of executed tests, respectively. Columns *SUCCESS RATE* present the number of executed tests per app in percentage. The ignored tests are those marked with the tag *@Ignore* and did not execute. In the present study, a test failed to execute due to a compilation error (Column IGN_C), an execution error (Column IGN_E), or flakiness (Column IGN_F).

Table 46 – Statistics of tests generated per app.

App	Statistics of Executed Tests				Statistics of Ignored Tests		
	N_{TEST}	N_{IGN}	N_{EXEC}	SUCCESS RATE (%)	IGN_C	IGN_C	IGN_F
A2DP	41	5	36	87.80%	0	3	2
AOBD	110	50	60	54.55%	1	41	8
ASCL	290	30	260	89.66%	0	27	3
AUTH	225	14	211	93.78%	0	14	0
C2T	216	46	170	78.70%	5	37	4
GB	389	26	363	93.32%	1	15	10
GPSL	551	45	506	91.83%	3	41	1
RU	160	31	129	80.63%	1	30	0
ULOG	188	28	160	85.11%	6	21	1
WS	108	15	93	86.11%	0	15	0
Total	2278	290	1988	87.27%	17	244	29
		12.73%	87.27%	84.43%	5.86%	84.14%	10.00%

The total number of tests generated for the ten apps is 2278, an approximate average of 228 tests per app. The GPSL application comprises the most tests, whereas A2DP has the fewest. A plausible explanation for the difference in the number of tests generated for the A2DP application compared to the others was that only a single class of the entire application was considered. Additionally, the information regarding the number of tests generated, ignored, and executed by each class is provided in Appendix B.1.

We observed that the majority of the tests (87.27%) generated by LLM were successfully executed. On the other hand, 84.14% of the ignored tests compiled but failed to execute correctly (e.g., assertion errors). 10% of tests were ignored due to flakiness, and only 5.86% were ignored due to compilation errors.

Table 47 presents the number of tests generated for each LLM.

Table 47 – Number of tests per LLM.

LLM	N_{TEST}	N_{IGN}	N_{EXEC}	SUCCESS RATE (%)
Claude	1153	115	1038	90.03%
DeepSeek	382	55	327	85.60%
ChatGPT	331	28	303	91.54%
Qwen	412	92	320	77.67%

As can be seen, Claude comprises over 50% (1153) of all generated tests (2278). On the other hand, DeepSeek (382) and ChatGPT (331) generated the fewest tests, with averages of 38.2% and 33.1% per app, respectively. Considering the ignored tests, Qwen had the most significant rate of discarded tests (22.3%), followed by DeepSeek (14%). In general, the two paid LLMs generated more useful tests than the free ones.

Moreover, Qwen had the highest failure rate (9.78%), whereas in ChatGPT, all tests compiled successfully. DeepSeek generated the most significant number of flaky tests (16.36%), followed by Qwen (10.87%).

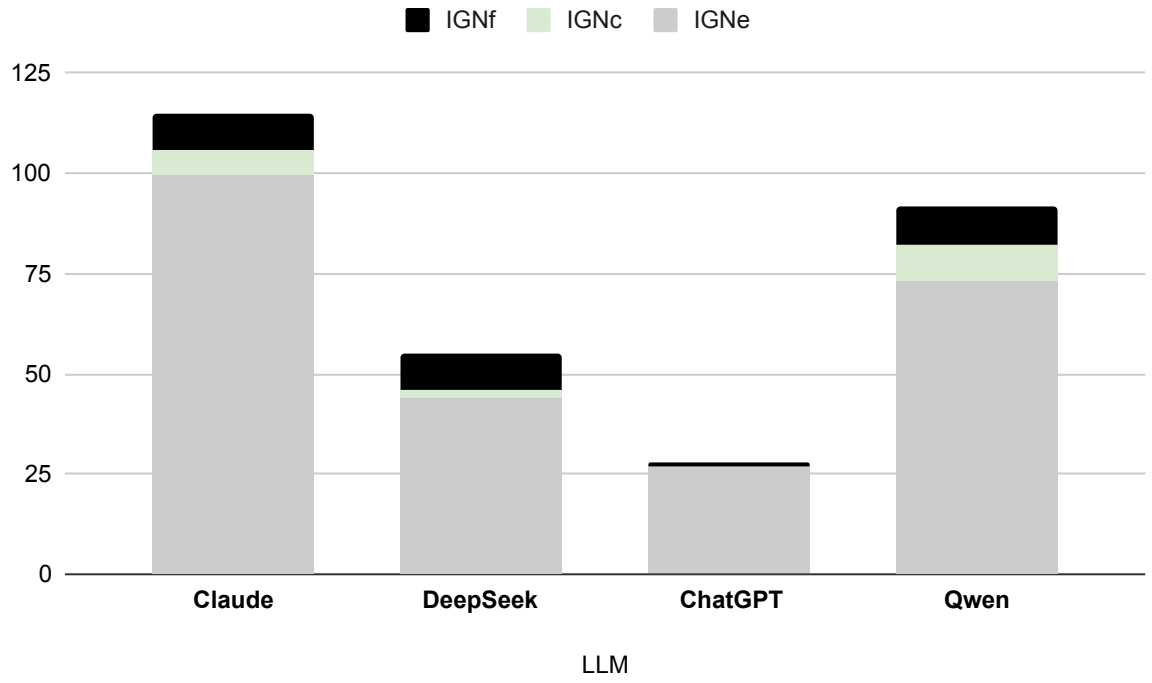


Figure 53 – Number of ignored tests per LLM

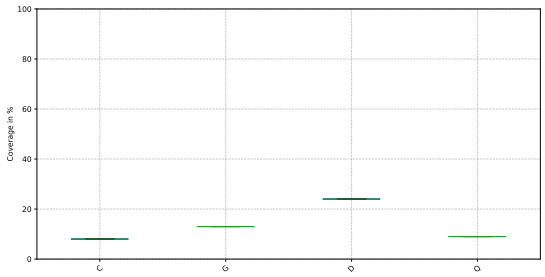
6.5.2 Code Coverage – Results

In this section, we present the results regarding code coverage of the tests generated by the LLMs. For space reasons, we presented only information on statement coverage. The data on branch coverage is also made available in the provided repository.

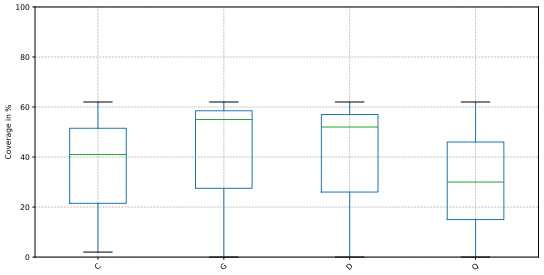
Figure 54 summarizes the results of the code coverage of each app through a boxplot chart. The y-axis shows coverage as a percentage on a scale of 0 to 100, and the x-axis shows results for Claude (C), ChatGPT (G), DeepSeek (D), and Qwen (Q).

Initially, we observed that the tests from Claude had better results regarding code coverage. In many cases, the test sets from this LLM showed more stable performance across the ten apps. Additionally, Claude had more outliers compared to the others, showing that, despite achieving interesting overall results, there were still some classes that the LLM failed to cover (e.g., Figure 54(e), Figure 54(g), and Figure 54(h)).

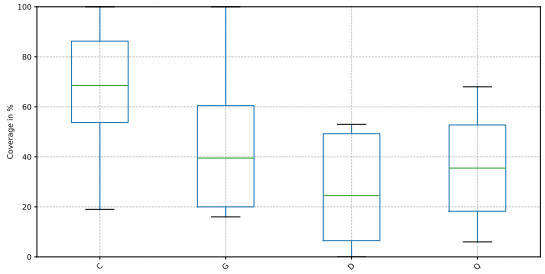
ChatGPT also achieved more interesting code coverage results than Claude. The main difference was that the first had greater coverage dispersion across the classes than the



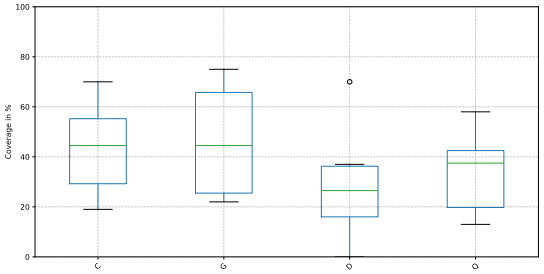
(a) Coverage – A2DP



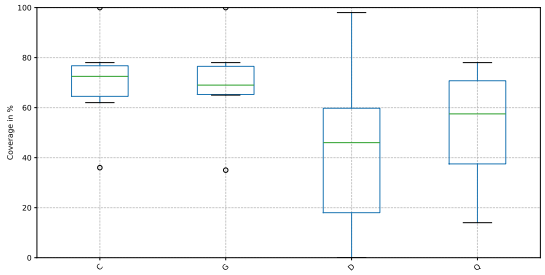
(b) Coverage – AOBD



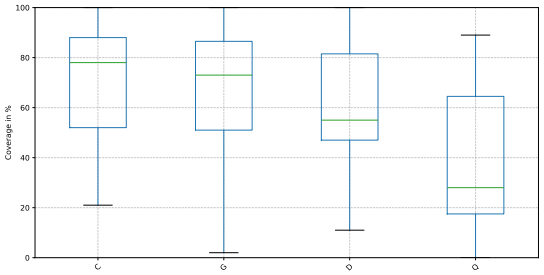
(c) Coverage – ASCL



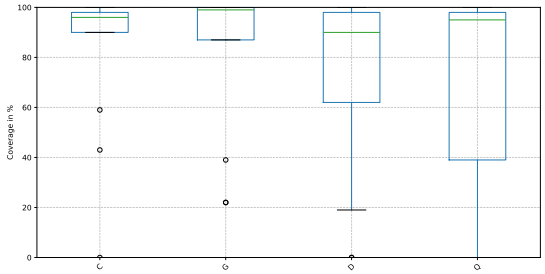
(d) Coverage – AUTH



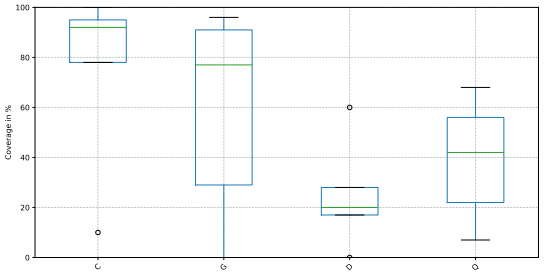
(e) Coverage – C2T



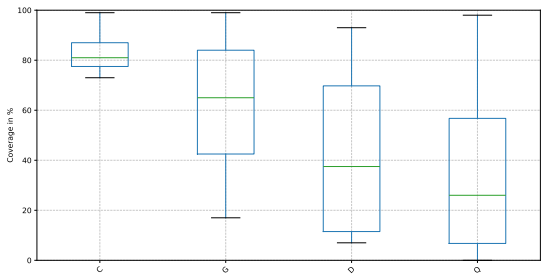
(f) Coverage – GB



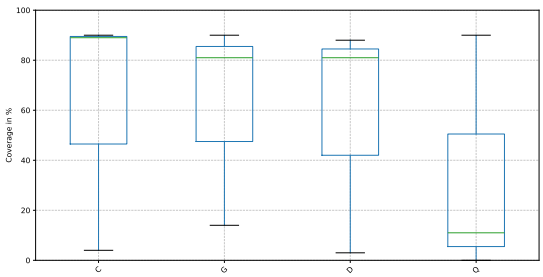
(g) Coverage – GPSL



(h) Coverage – RU



(i) Coverage – ULOG



(j) Coverage – WS

Figure 54 – Code coverage for all subject apps.

latter. The classes from DeepSeek and Qwen performed the least. In many cases, the median of these two LLMs was below the median of Claude and ChatGPT, suggesting lower coverage among the LLMs.

Next, the code coverage of each LLM for each app is presented in Table 48. For a fair comparison, the final coverage metric was computed as the average code coverage across the selected classes (see Table 42).

Table 48 – Coverage score of all LLMs in %.

App	Claude	ChatGPT	DeepSeek	Qwen
A2DP	8	13	24	9
AOBD	35	39	38	30.67
ASCL	66.17	46	26.67	36
AUTH	43.5	46.33	29.17	34
C2T	70.17	69.33	43.83	52.17
GB	69.45	65.18	60.82	39.82
GPSL	81.85	81.15	71.62	72.69
RU	75	58.6	25	39
ULOG	83.5	61.5	43.75	37.5
WS	61	61.67	57.33	33.67
AVG	59.36	54.18	42.02	38.45

Claude achieved the highest average code coverage of 59.36%, followed by ChatGPT (54.18%) and DeepSeek (42.02%). Qwen had the worst coverage performance, with 38.45%, approximately 21% lower than Claude. The classes from GPSL achieved higher code coverage than the other apps, whereas A2DP had the lowest coverage.

To validate the results, we performed a statistical test to assess differences in code coverage among the groups. Therefore, we defined the following hypotheses:

- **Null Hypothesis (H_0):** There is no difference in the code coverage among the LLMs.
- **Alternative Hypothesis (H_1):** There is a difference in the code coverage among the LLMs.

The process began with the Shapiro-Wilk test to assess the normality of the samples at the 95% confidence level (i.e., a baseline *p-value* of 0.05). This analysis considered the total code coverage of each app as presented in Table 48. According to Figure 55, the samples were normally distributed. Thus, the Student's t-test was used to assess differences among the groups at the 95% confidence level.

From Figure 56, the pair (Claude; Qwen) had a resulting p-value lower than the baseline, showing a statistical difference between these two groups. Therefore, H_0 was rejected; hence, we can infer that there was a statistical difference in the code coverage of the tests generated by Claude and Qwen.

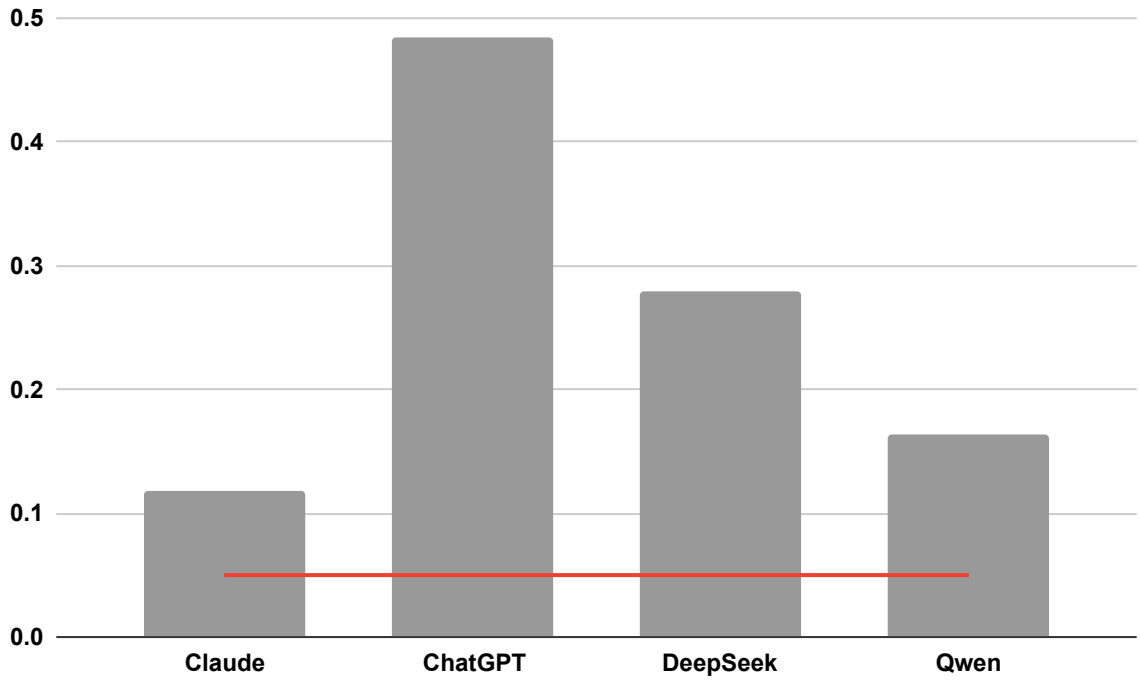


Figure 55 – Results of Shapiro-Wilk test – Code coverage.

Considering the other pairs, we observed that the p-value was larger than the baseline. In these cases, we rejected the alternative hypothesis (H_1), inferring that there was no statistical difference in code coverage between these pairs.

Next, we applied Cohen's d effect size test to verify whether a sample achieves larger code coverage than another sample. The general guidelines for interpreting the results of the effect size are as follows:

- ❏ Small effect: 0.2
- ❏ Medium effect: 0.5
- ❏ Large effect: 0.8

According to Figure 57, there was a large effect size comparing Claude to DeepSeek and Qwen, and ChatGPT to Qwen, showing that the code coverage among these samples was different, i.e., Claude has a larger code coverage compared to DeepSeek and Qwen, and ChatGPT had a larger coverage compared to Qwen.

Considering ChatGPT and DeepSeek, we observed a medium-to-large effect size. Thus, we can infer that there was a difference in the coverage considering this sample. The other two samples had a small effect, indicating no statistical difference in the final code coverage.

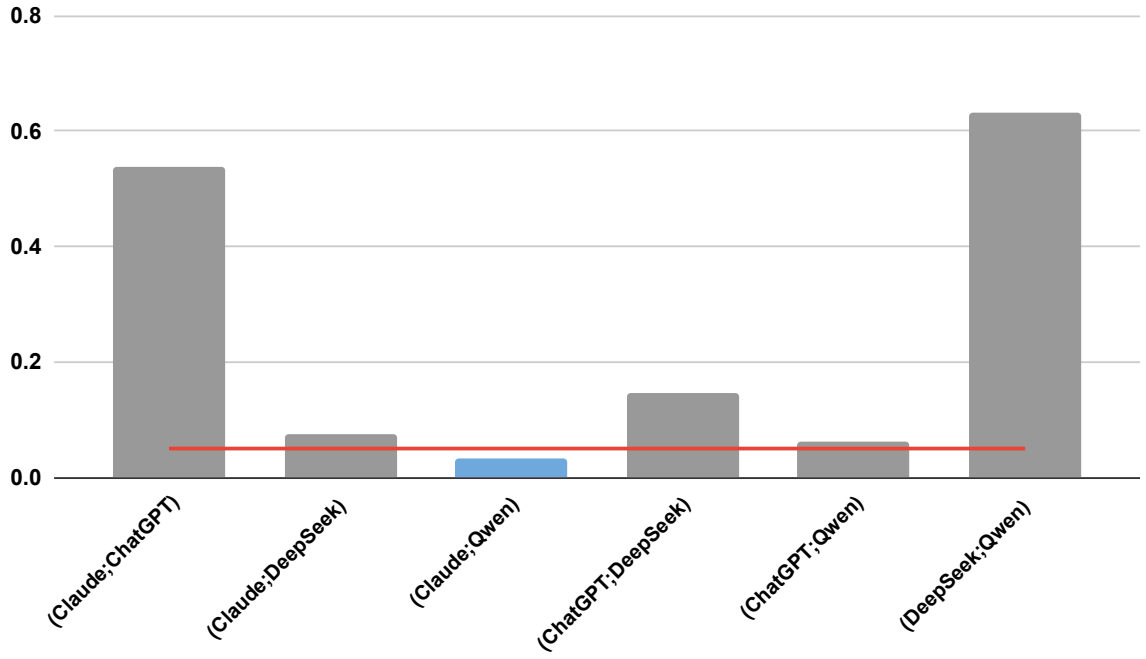


Figure 56 – Results of Student's t-test – Code coverage.

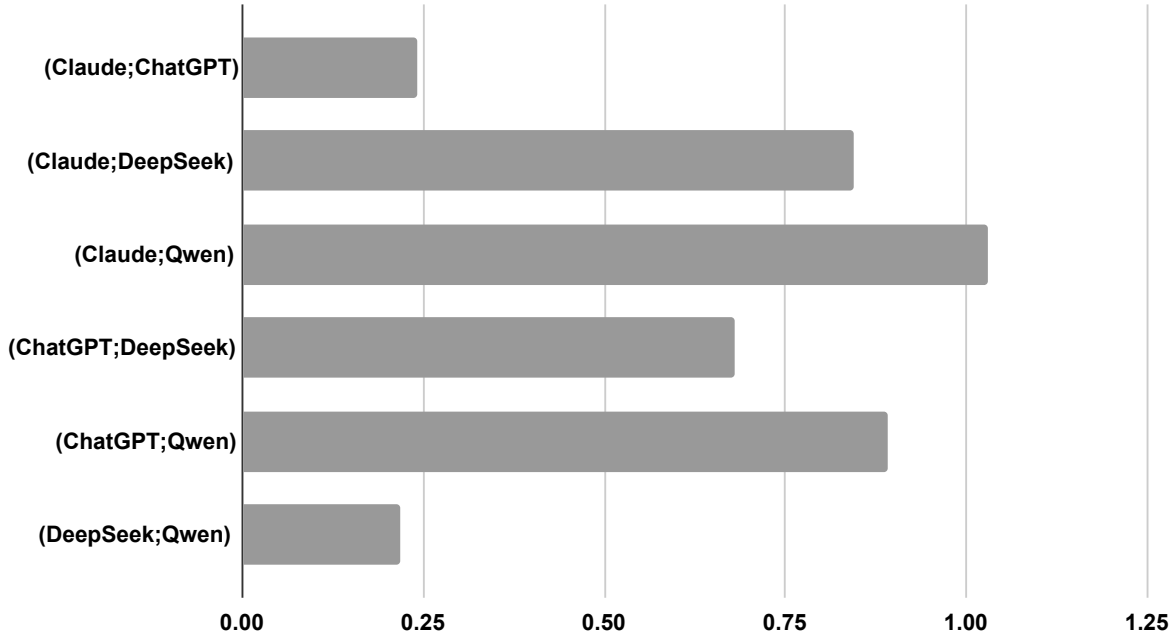


Figure 57 – Results of Cohen's d Effect Size – Code coverage.

6.5.2.1 Combining Test Cases from LLMs – Code Coverage

In this section, we investigated the effects on code coverage by combining the test sets of the LLMs. That is, we assessed the extent of code coverage evolution when combining test sets from different sources. To this end, the following pairing combinations were considered:

- ❏ C+G: Tests from Claude and ChatGPT.
- ❏ C+D: Tests from Claude and DeepSeek.
- ❏ C+Q: Tests from Claude and Qwen.
- ❏ G+D: Tests from ChatGPT and DeepSeek.
- ❏ G+Q: Tests from ChatGPT and Qwen.
- ❏ D+Q: Tests from DeepSeek and Qwen.

Table 49 presents the results , and Figure 58 compares the paired combinations against

Table 49 – Code coverage of paired combinations in %.

App	C+G	C+D	C+Q	G+D	G+Q	D+Q
A2DP	15	25	12	31	17	25
AOBD	39.67	38.67	35	39	39	38
ASCL	71	74.67	82.5	56	62.33	39.67
AUTH	56.33	46.17	51	50.67	50.67	38.5
C2T	78	70.17	72.83	69.83	74.33	62.83
GB	79.45	74	69.45	72.73	69.09	60.91
GPSL	84.85	82.38	82.08	82.77	81.46	78.77
RU	79.6	77.6	75	63.4	63	43
ULOG	84.5	84.25	83.75	61.75	62.25	45.75
WS	64.67	61.33	63.67	62.33	64.67	61
AVG	65.31	63.42	62.73	58.95	58.38	49.34

individual coverage. As shown, combining the test sets increased code coverage compared to running them individually. For instance, combining C+G achieved a higher score (65.31%) than Claude (59.36%) and ChatGPT (54.18%); C+D (63.42%) achieved a higher score than Claude (59.36%) and DeepSeek (42.02%), and so on. Hence, this data shows that the test sets generated by the LLMs exhibit some degree of complementarity. That is, the generated test cases covered distinct parts of the source code. On the other hand, G+D (58.85%), G+Q (58.38%), and G+D (49.34%) increase code coverage; however, these combinations could not outperform Claude individually.

Next, we estimated the average coverage gain across paired samples to assess the extent of their complementarity. The metric was computed as the average coverage gain per app. Figure 59 illustrates the results.

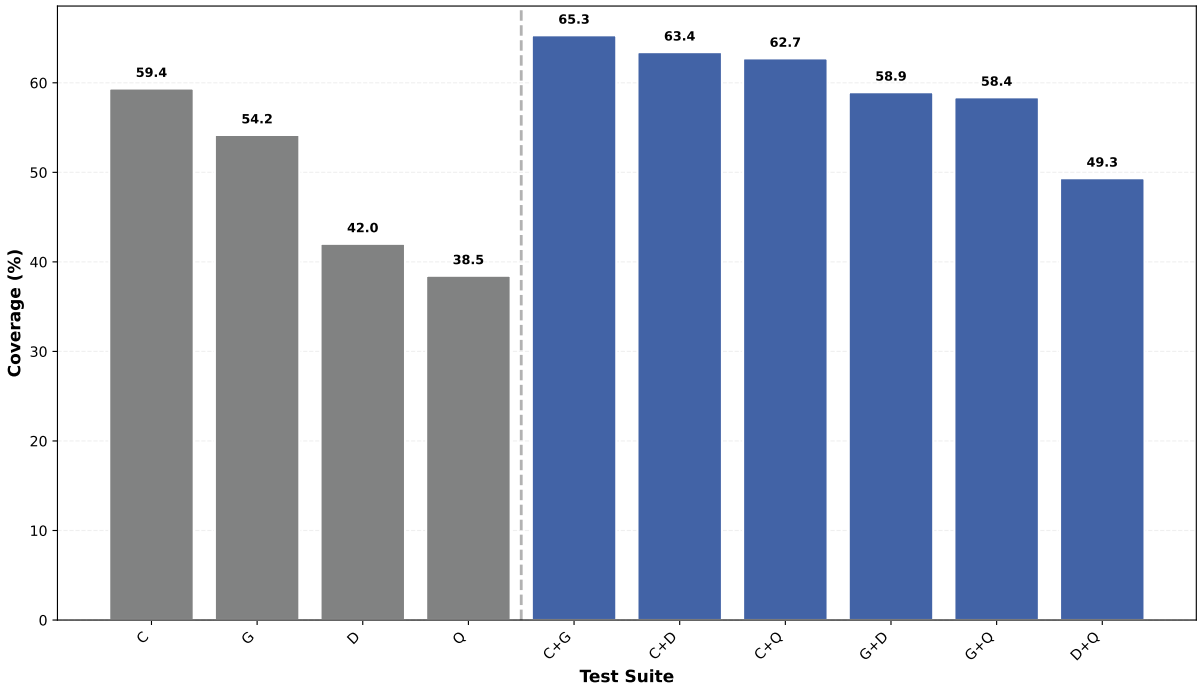


Figure 58 – Comparison of paired test sets – Code coverage.

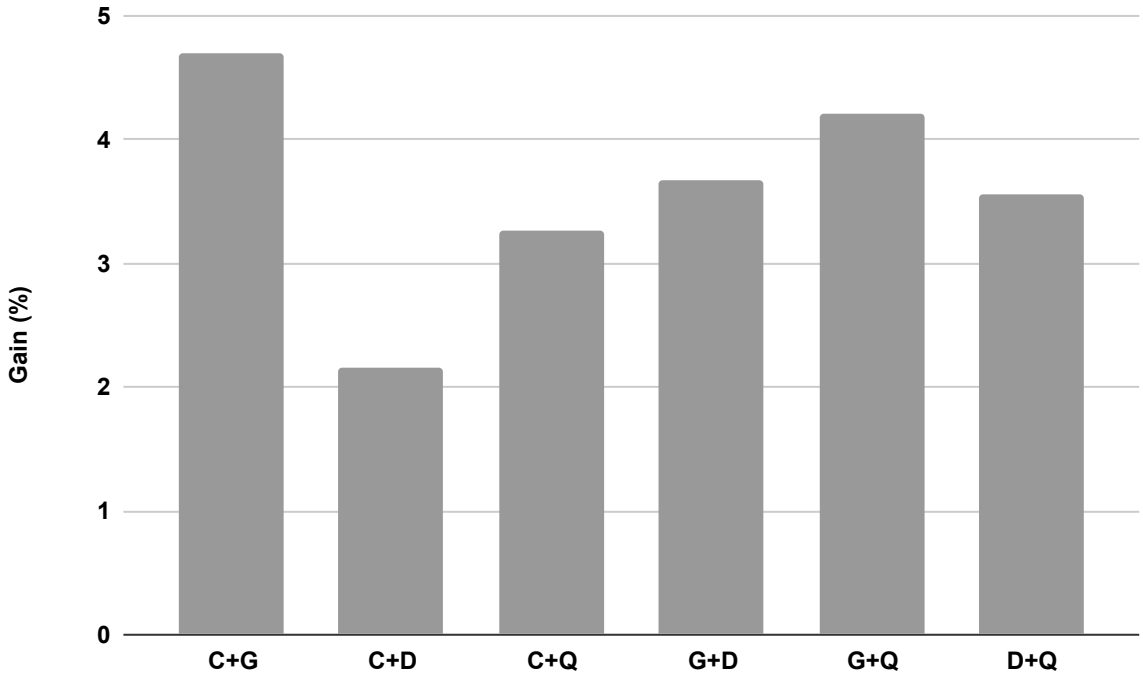


Figure 59 – Code coverage gain among paired samples.

Overall, C+G had the greatest coverage gain (4.69%) among the samples, followed by G+Q and G+D, respectively. The sample C+D achieved the lowest coverage gain

of 2.16%. In other words, the tests generated from DeepSeek had little influence when combined with Claude. Moreover, we observed that the Qwen test set showed significant coverage complementarity. Despite having the lowest overall coverage, Qwen's tests may exercise other parts of the code that the other LLMs did not.

Finally, we analyze code coverage incrementally. That is, we investigated the behavior of the code coverage by combining the test sets incrementally in ascending and descending order of coverage:

- ❏ Descending (D): Claude → ChatGPT → DeepSeek → Qwen.
- ❏ Ascending (A): Qwen → DeepSeek → ChatGPT → Claude.

Table 50 summarizes the findings, and Figure 60 illustrates the incremental code coverage growth rate. It was possible to visualize the significant impact of Claude on

Table 50 – Incremental code coverage in %.

Descending	
LLM	Coverage
C	59.36
C+G	65.31
C+G+D	67.66
C+G+D+Q	69.18

Ascending	
LLM	Coverage
Q	38.45
Q+D	49.34
Q+D+G	60.89
Q+D+G+C	69.18

the final code coverage. When combining Qwen, DeepSeek, and ChatGPT (60.89%), the code coverage was slightly better than Claude (59.36%). ChatGPT also made a significant contribution. In the first case, the coverage score increased from 59.36% to 65.31% and, in the second case, from 49.34 to 60.89%.

In summary, Claude and ChatGPT were the most effective in terms of code coverage. Observing Figure 60, the strategy $C \rightarrow Q$ had marginal gains after combining Claude and ChatGPT, whereas $Q \rightarrow C$ had a considerable gain after combining the tests from these LLMs.

6.5.3 Mutation Testing – Results

This section provides the results of mutation testing. For this purpose, we evaluated the quality of the test cases generated by each LLM based on its fault-detection capabilities.

Figure 61 illustrates the number of mutants generated per application, which varies from three (app ULOG) to 21 (app ASCL), totaling 103 mutants with an average of 10.3 per app. At first glance, one might argue that the number of mutants was relatively low compared to that of traditional mutation operators. However, since we are dealing with a specific type of mutants, the number of mutants is somehow expected to be lower than traditional operators.

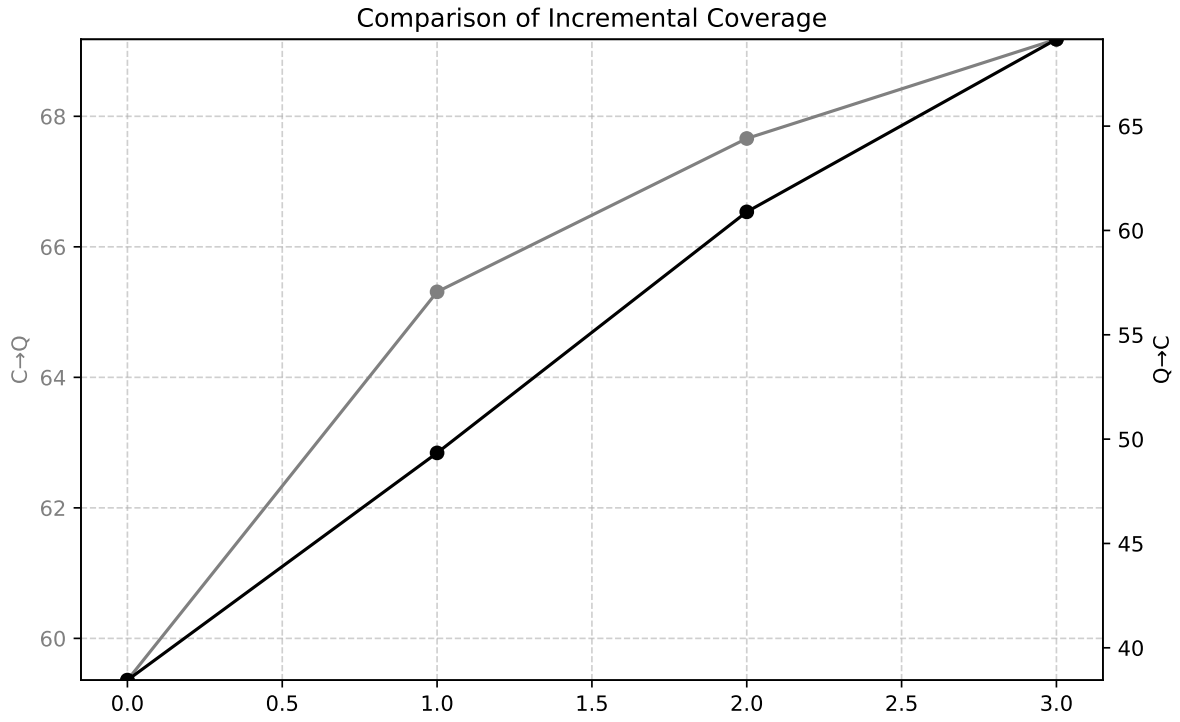


Figure 60 – Growth rate of incremental code coverage.

The same configuration file was also set for all applications (see Figure 51). This may limit the number of mutants depending on their code structure.

Table 51 presents information regarding the number of mutants generated by each mutation operator. As observed, the NDID operator generated the most mutants (76), followed by DDM (15). On the other hand, the mutation operators BDC, DDBID, RLPDR, RLRBPDR, and TDC did not generate any mutants for this set of applications.

Table 51 – Number of mutants generated per mutation operator.

App	**BDC**	**BDL**	**DDBID**	**DDM**	**NDID**	**RAIDD**	**RLPDR**	**RLRBPDR**	**TDC**	**TDL**	Total
A2DP	0	0	0	0	4	1	0	0	0	0	5
AOBD	0	0	0	0	6	1	0	0	0	0	7
ASCL	0	0	0	0	18	0	0	0	3	0	21
AUTH	0	0	0	0	8	0	0	0	0	0	8
C2T	0	2	0	2	7	2	0	0	1	0	14
GB	0	0	0	4	9	0	0	0	0	0	13
GPSL	0	0	0	0	6	0	1	0	0	0	7
RU	0	0	0	3	11	0	4	0	0	0	18
ULOG	0	0	0	1	2	0	0	0	0	0	3
WS	0	0	0	0	5	0	0	0	2	0	7
Total	0	2	0	10	76	4	5	0	6	0	**103**

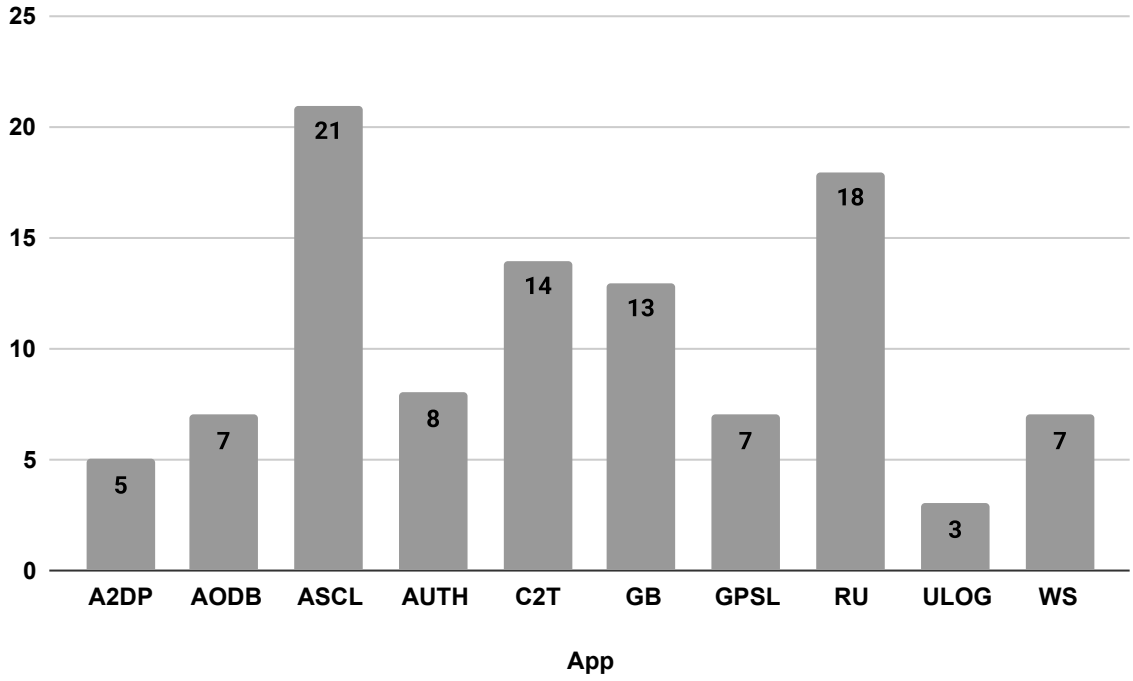


Figure 61 – Number of mutants generated per app.

Next, Table 52 presents the mutation score of each LLM. Columns $\mathbf{MS_C}$, $\mathbf{MS_G}$, $\mathbf{MS_D}$, and $\mathbf{MS_Q}$ display the mutation score of Claude, ChatGPT, DeepSeek, and Qwen, respectively, for each application. Column $\mathbf{MS_T}$ presents the mutation score by combining the tests of all four LLMs. The cells highlighted in gray represent the LLM's highest mutation score for a given app. The cell highlighted in light blue depicts the cases where the mutation score increased after combining all tests.

Table 52 – Mutation score for each LLM in %.

App	$\mathbf{MS_C}$	$\mathbf{MS_G}$	$\mathbf{MS_D}$	$\mathbf{MS_Q}$		$\mathbf{MS_T}$
A2DP	20.00	0.00	0.00	0.00		20.00
AOBD	28.57	57.14	28.57	28.57		71.43
ASCL	9.52	0.00	38.10	42.86		52.38
AUTH	75.00	75.00	62.50	75.00		75.00
C2T	57.14	35.71	14.28	57.14		64.28
GB	46.15	30.77	15.38	15.38		46.15
GPSL	28.57	28.57	28.57	28.57		28.57
RU	27.78	16.67	5.56	0.00		33.33
ULOG	33.33	33.33	66.67	33.33		66.67
WS	85.71	0.00	42.86	0.00		85.71
Avg	41.18	27.72	30.25	28.09		54.35

Overall, Claude obtained the largest average mutation score among the LLMs, 41.18%. Additionally, Claude outperformed the other LLMs across four of ten apps: A2DP, GB, RU, and WS. DeepSeek had the best mutation score for the ULOG app. ChatGPT scored

best for the AOBD app, whereas Qwen outperformed the other LLMs in ASCL. For the AUTH app, Claude, ChatGPT, and Qwen achieved the same mutation score of 75%. In C2T, Claude and Qwen had the highest scores, and in GPSL, all four LLMs had killed the same number of mutants.

For the AUTH application, ChatGPT achieved the same score with the smallest test size (43) compared to Claude (83) and Qwen (58). For C2T, the difference in test set size is even more considerable. Qwen achieved the same score as Claude, with 42 tests, compared to 110 for the latter. For GPSL, ChatGPT achieves the same score as Claude (256), DeepSeek (112), and Qwen (105) across 78 tests.

Considering column MS_T, in four apps, the mutation score increased after combining the test set: AOBD, ASCL, C2T, and RU. In the other six apps, we did not observe any difference in the final mutation score, indicating that, despite improving the test sets from different sources, there is no evidence of complementarity among them; i.e., they kill the same subset of mutants.

Next, we performed statistical testing to support the findings better. To this end, we aim to investigate whether there is a statistical difference between the samples, i.e., the mutation scores for each LLM. Therefore, the following hypotheses were defined:

- **Null Hypothesis (H_0):** There is no difference among the mutation scores of each LLM.
- **Alternative Hypothesis (H_1):** There is a difference among the mutation scores of each LLM.

Once again, the Shapiro-Wilk normality test was applied to determine whether the sample had a normal distribution, considering a confidence level of 95% (i.e., baseline *p-value* of 0.05). Figure 62 displays information about the obtained p-values. The sample comes from a normal distribution since p-values were higher than the baseline.

Given the normality test results, we applied the Student's t-test to verify whether there was a statistical difference between the samples. For this analysis, we conducted a pairwise comparison among the LLMs at the 95% confidence level. Figure 63 displays the results.

As can be observed, in all six cases, the resulting p-values were larger than the baseline, and hence, the alternative hypothesis (H_1) was rejected. This shows that there was no statistical difference among the mutation scores obtained by the LLMs.

Finally, Table 52 indicates that the mutation score of Claude (MS_C) may be higher than the others. Therefore, we also performed Cohen's d effect size test to determine whether a sample of mutation scores was superior to another.

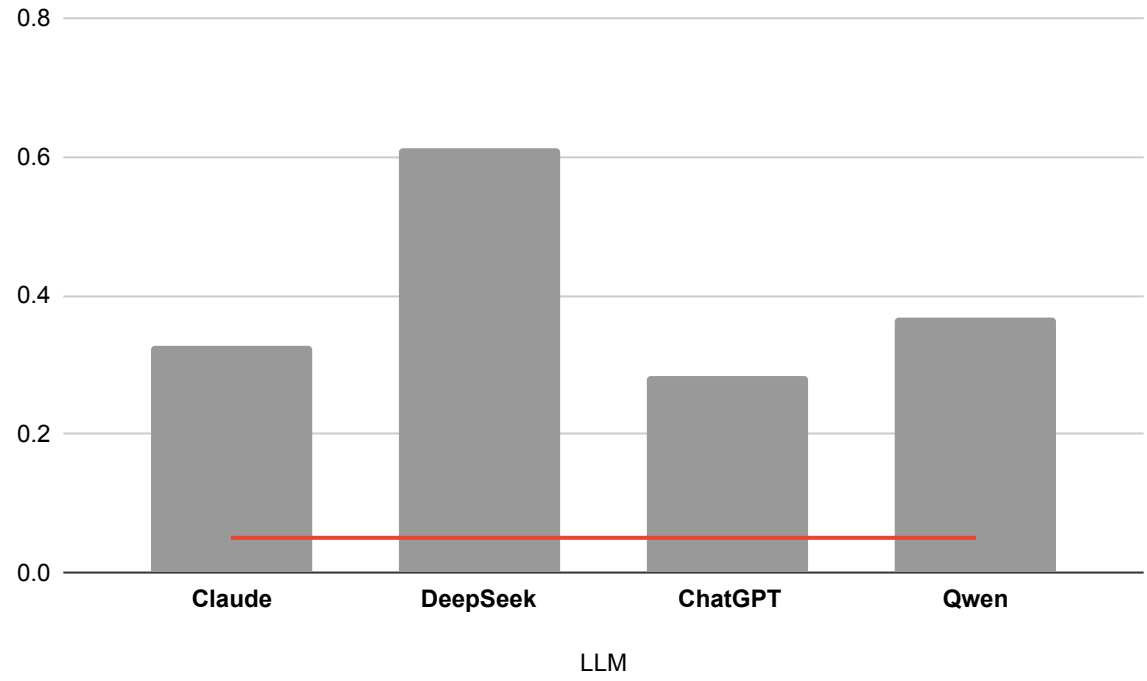


Figure 62 – Results of Shapiro-Wilk test – Mutation testing.

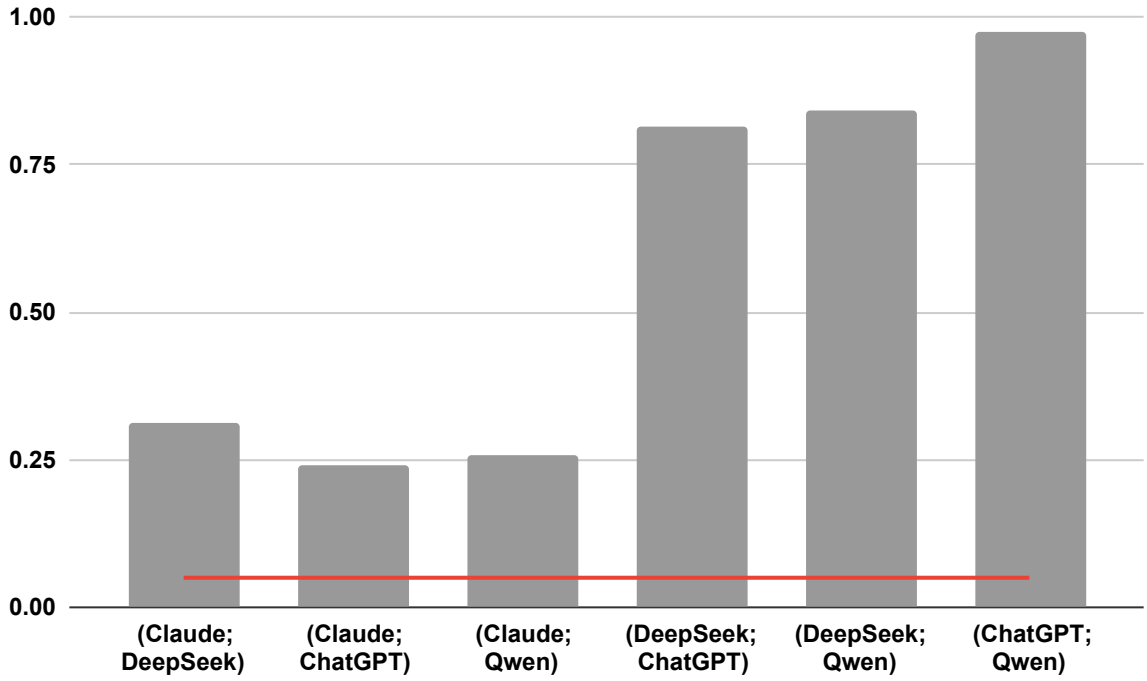


Figure 63 – Results of Student's t-test – Mutation testing.

Figure 64 presents the results of Cohen's d effect size test. As can be observed, the pairs (DeepSeek; ChatGPT), (DeepSeek; Qwen), and (ChatGPT; Qwen) have a small effect size, which there is no substantial difference between these samples, meaning that, for these cases, an LLM did not outperform the other in terms of mutation score.

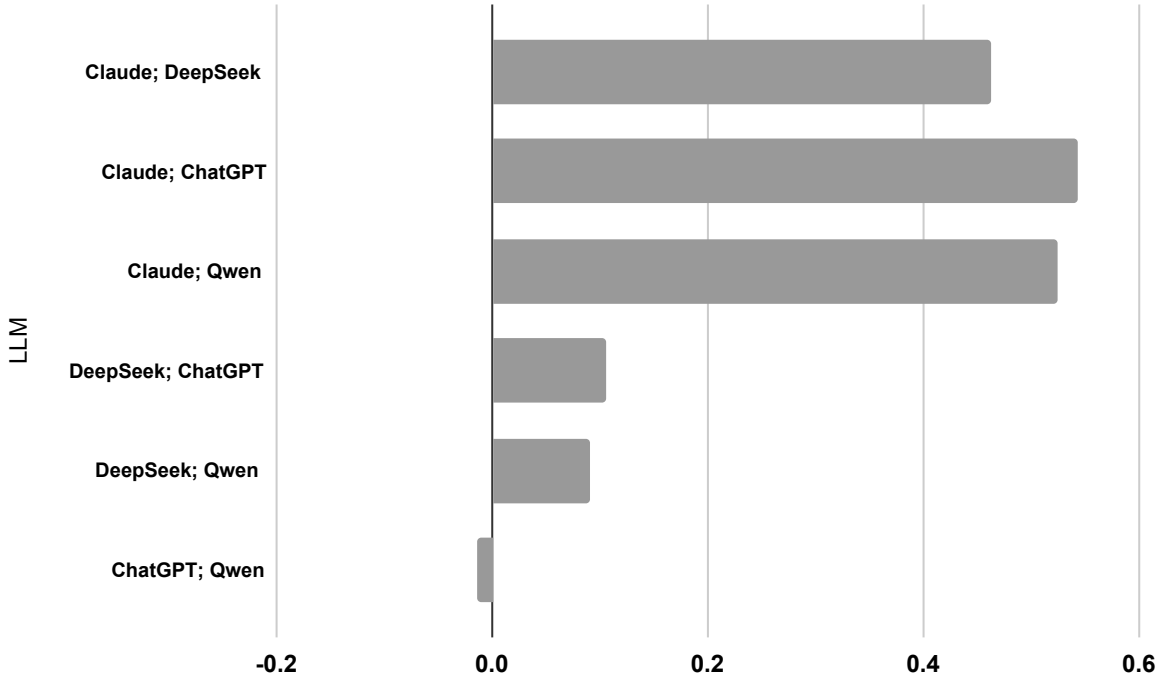


Figure 64 – Results of Cohen's d effect size – Mutation testing.

On the other hand, the results comparing Claude to the other LLMs show a small-to-medium effect size (0.46) between Claude and DeepSeek. Moreover, medium-to-large effect sizes were observed when comparing Claude to ChatGPT (0.54) and Claude to Qwen (0.52). Therefore, it is possible to infer a difference in Claude's mutation score compared to DeepSeek, ChatGPT, and Qwen.

Note that, despite Cohen's d effect size test indicating that Claude may outperform the other three LLMs in fault detection, we argue that the results may not be generalizable across all contexts due to the sample size used in the experimental study.

Finally, we present the results of the time spent running the tests against the mutants. Table 53

The time spent for a single run was approximately 2.250 minutes (i.e., 37 hours and 30 minutes). Since mutation testing was run 5 times, the total time spent in the experimental study was 187 hours and 55 minutes (i.e., 7 days, 19 hours, and 55 minutes).

Note that the number of mutants generated for these 10 apps was lower than with traditional mutation operators. Also, the tests were generated at the unit level, which tends to execute faster than GUI tests.

Table 53 – Time spent to run mutation testing.

App	$T_{MINUTES}$	T_{HOUR}
A2DL	19	0h 19m
AOBD	54	0h 54m
ASCL	232	3h 52m
AUTH	261	4h 21m
C2T	261	4h 21m
GB	637	10h 37m
GPSL	320	5h 20m
RU	289	4h 49m
ULOG	59	0h 59m
WS	118	1h 58m
Total	2250	37h 30m
Avg	225	3h 45m

Therefore, running Android instrumented tests on a real device tends to be more time-consuming. This can be explained by the need to send commands to the device via *adb*, similar to a client-server architecture. Since these Android components (e.g., Bluetooth and Location) require running tests on real devices due to emulator constraints, we advocate treating the time spent running tests for these components as an essential variable to consider. Observe that execution time can be sped up by parallel execution across multiple devices, whether physically connected to the computer or remotely, using a platform like Distributed Bug Buster (DBB) (FARIA et al., 2019).

6.5.3.1 Combining test cases from LLMs – Mutation testing.

In this section, we conducted the same investigation presented in Section 6.5.2.1, but with a focus on mutation score. Table 54 presents the results of pairing the test sets of the LLMs.

Table 54 – Mutation score of paired combination in %.

App	C+G	C+D	C+Q	G+D	G+Q	D+Q
A2DP	20	20	20	0	0	0
AOBD	71.43	42.86	28.57	57.14	71.43	42.86
ASCL	9.52	42.86	47.62	38.1	42.86	47.62
AUTH	75	75	75	75	75	75
C2T	57.14	57.14	64.28	35.71	57.14	57.14
GB	46.15	46.15	46.15	38.46	30.77	23.08
GPSL	28.57	28.57	28.57	28.57	28.57	28.57
RU	27.78	33.33	27.78	22.22	16.67	5.6
ULOG	33.33	66.67	33.33	66.67	33.33	66.67
WS	85.71	85.71	85.71	42.86	0	42.86
AVG	45.46	49.83	45.70	40.47	35.58	38.94

As shown, the average mutation score increased across all paired combinations compared to their individual samples, indicating a degree of complementarity. On the other

hand, there is still a significant difference between the best pair C+G (45.45%) and the total mutation score (54.35%). These show that the LLMs generated tests for different parts of the source code and can detect a different set of faults.

When assessing the mutation score gain, few apps increased their score after pairing. Observing Table 55, only half of the apps had a certain gain level. For instance, AOBD increased the mutation in four of six pairing combinations.

Table 55 – Mutation score gain per app in %.

App	C+G	C+D	C+Q	G+D	G+Q	D+Q
A2DP	0	0	0	0	0	0
AOBD	14.29	14.29	0	0	14.29	14.29
ASCL	0	4.76	4.76	0	0	4.76
AUTH	0	0	0	0	0	0
C2T	0	0	7.15	0	0	0
GB	0	0	0	7.69	0	7.7
GPSL	0	0	0	0	0	0
RU	0	5.55	0	5.55	0	0
ULOG	0	0	0	0	0	0
WS	0	0	0	0	0	0

Regarding average gain, the pairs D+Q and C+D had the highest gain percentages, respectively, whereas C+Q had the lowest. The overall data suggest that DeepSeek had a significant complementary level when combined with the other LLMs. Figure 65 summarizes the results.

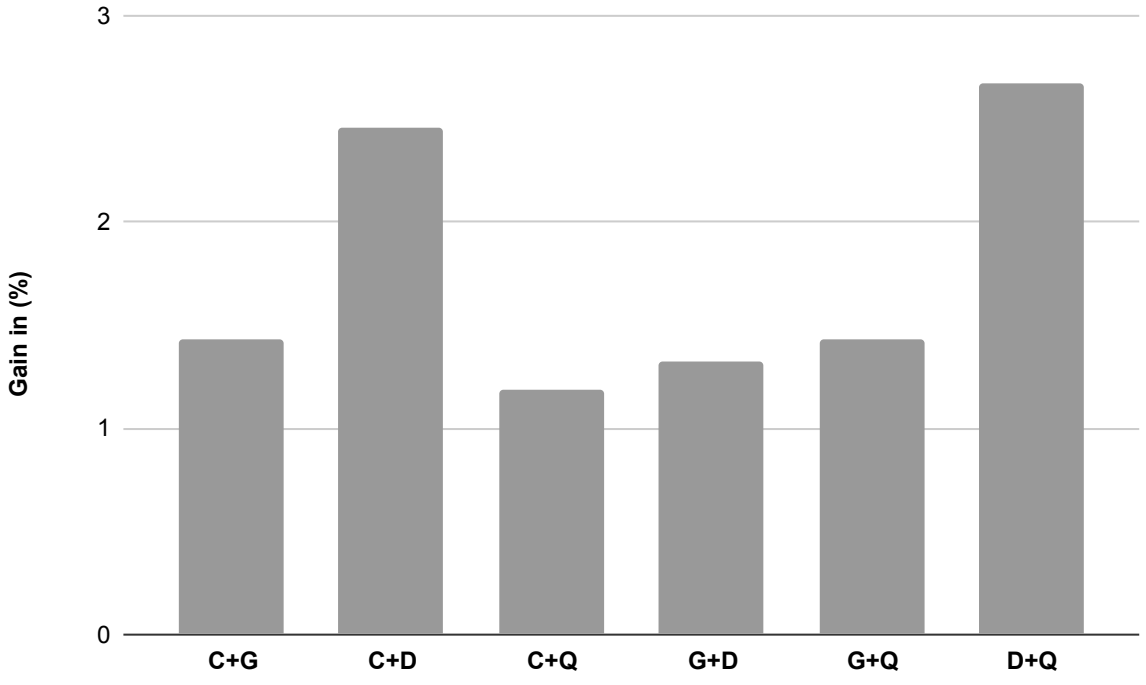


Figure 65 – Mutation score gain among paired samples.

Finally, Figure 66 presents the growth rate of the incremental mutation score. It is possible to observe the impacts of DeepSeek combined with the LLMs. Note that Claude had the highest mutation score among the participants. However, when combining the latter with DeepSeek, the mutation score surpassed C+G. Next, when combining DeepSeek with C+G, the mutation score surpassed Q+D+C. This provides strong evidence that the DeepSeek test sets killed more mutants than those from Claude, Qwen, and ChatGPT.

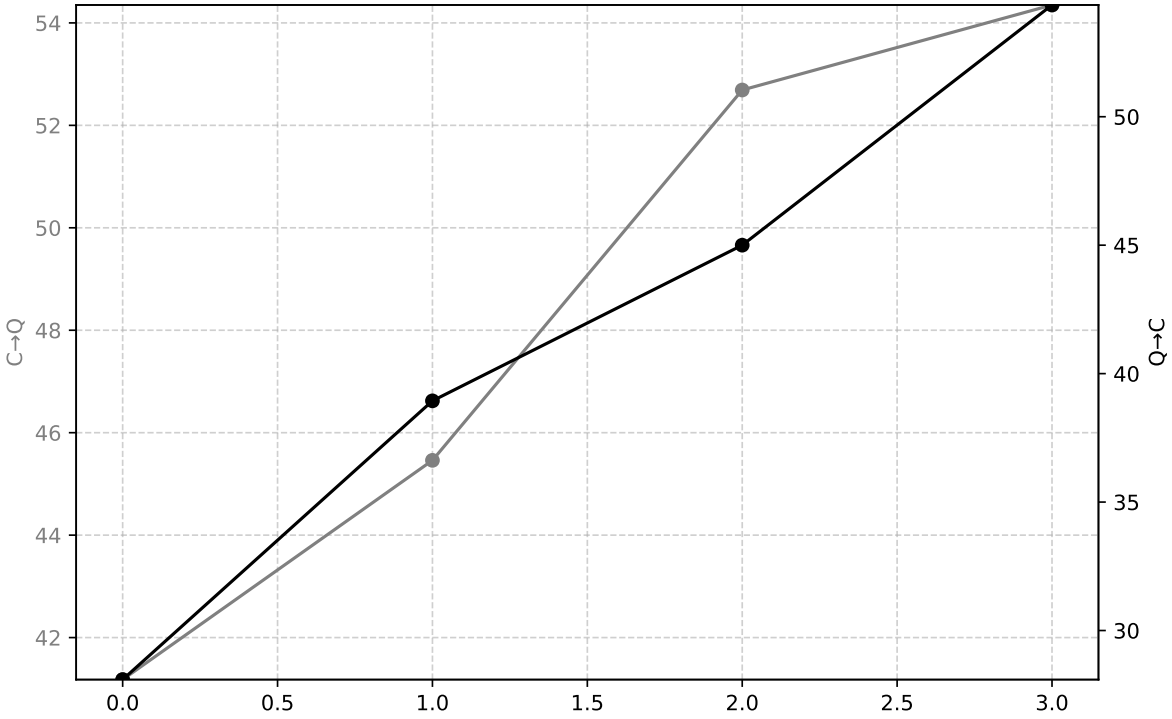
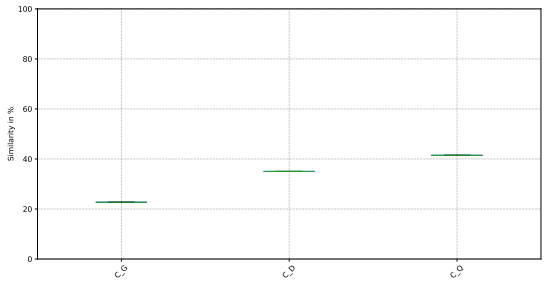


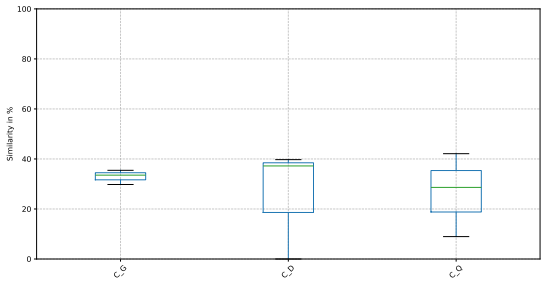
Figure 66 – Growth rate of incremental mutation score.

6.5.4 chRF – Test Case Similarity

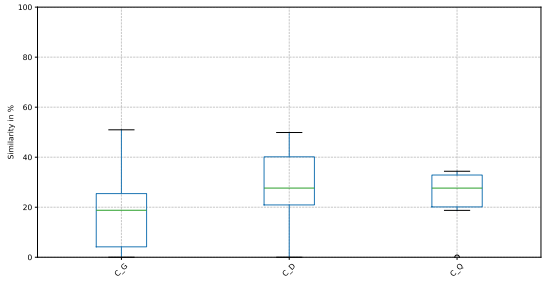
This section presents the results of the CHaRacter-level F-score (or chrF) to evaluate the similarity among the test codes generated by the LLMs. As discussed before, the chrF metric compares machine translation output and a reference translation. In the present study, we set the test generated by Claude as the reference, as it achieved better results than the other three LLMs. Figure 67 presents the results of the chrF metric for each application through a boxplot chart. The y-axis presents the percentage of similarity among the samples. In contrast, the x-axis presents the chrF score between Claude and DeepSeek, Claude and ChatGPT, and Claude and Qwen, respectively.



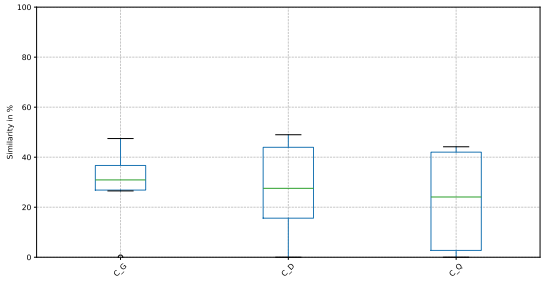
(a) chrF – A2DP



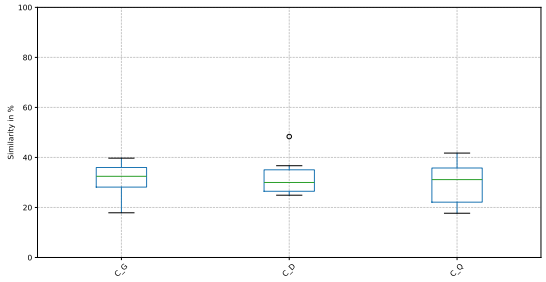
(b) chrF – AOBD



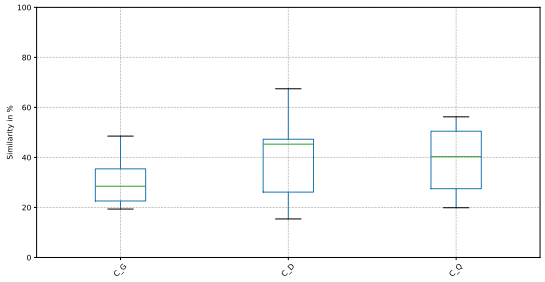
(c) chrF – ASCL



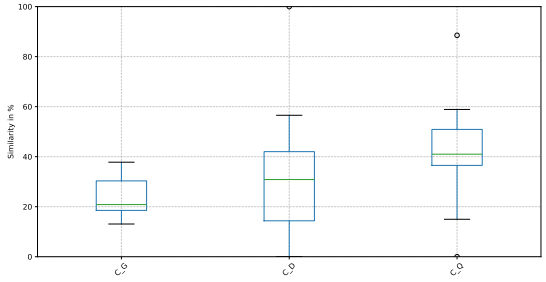
(d) chrF – AUTH



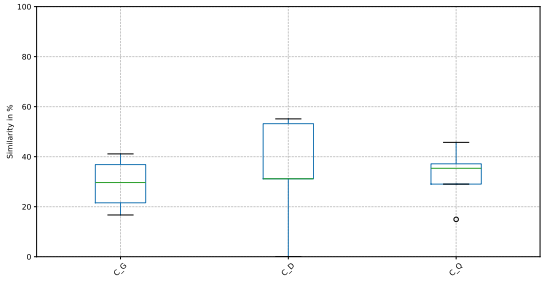
(e) chrF – C2T



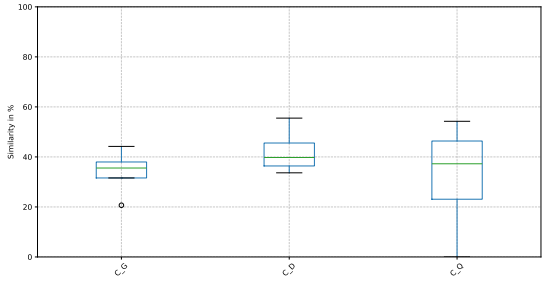
(f) chrF – GB



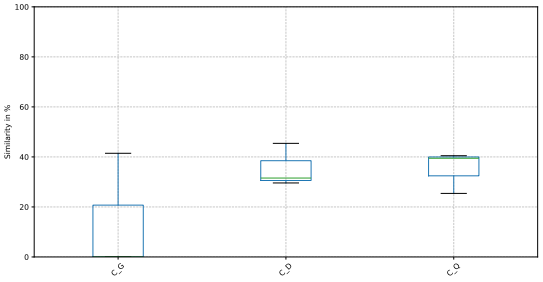
(g) chrF – GPSL



(h) chrF – RU



(i) chrF – ULOG



(j) chrF – WS

Figure 67 – chrF metric for each application.

The majority of test cases had average similarity scores below 50%, indicating little overlap between the test cases generated by the LLMs. On the other hand, some outliers were observed among the samples. The most remarkable is the score of class *GpxReader* among Claude and DeepSeek, which achieved 100% of similarity.

Table 56 presents the chrF similarity considering all tests generated by the LLMs of each app.

Table 56 – chrF score among all tests generated by LLMs.

App	Claude		
	ChatGPT	**DeepSeek**	**Qwen**
A2DP	22.68	35.02	41.45
AOBD	29.63	27.16	14.52
ASCL	22.56	25.3	21.58
AUTH	28.06	31.18	24.2
C2T	26.7	23.38	17.13
GB	26.69	27.82	25.11
GPSL	19.68	20.74	22.09
RU	18.83	21.79	18.28
ULOG	31.23	28.58	29.99
WS	38.99	27.86	32.8
AVG	26.50	26.88	24.71

The DeepSeek tests achieved the highest similarity score, followed by ChatGPT and Qwen. Comparing results against the mutation score (see Table 52, we observed that DeepSeek also had the largest mutation score compared to ChatGPT and Qwen. Moreover, ChatGPT had the second-largest chrF score and the lowest average mutation score.

To validate the results, we used Pearson's Correlation Coefficient to assess the correlation between chrF and mutation score, assuming normal distributions and a 95% confidence level. Table 57 presents the results.

Table 57 – Pearson's Correlation –chrF vs Mutation Score.

Pair	**r**	**p-value**
MS_G - $chrF_G$	0.105	0.772
MS_D - $chrF_D$	0.150	0.679
MS_Q - $chrF_Q$	-0.426	0.220

As can be observed, there is no meaningful correlation between the chrF and mutation score between ChatGPT and Deepseek, i.e., 0.105 and 0.150, respectively. Considering Qwen, there is a negative, moderate correlation (-0.426), but it is not statistically significant (p-value equal to 0.220).

Therefore, it is possible to infer that the LLMs produce different test sets, despite having similar fault-detection capabilities.

6.6 Mutation-Guided Test Case Generation

In the previous section, we evaluated the capabilities of LLMs (i.e., Claude, DeepSeek, ChatGPT, and Qwen) in generating tests for two Android device components: Bluetooth and Location. In this case, we started test generation from scratch, assuming the application lacked tests that exercised these components.

Now, we investigate the LLM's ability to generate tests that kill specific mutants that survived the previous stage. We called this strategy *mutation-guided test case generation*.

To guide this second experimental study, we defined the following research question:

RQ2: What is the capability of large language models in generating Android instrumented unit tests for Bluetooth and Location components, aiming at killing specific mutants?

The second experimental study started by identifying live mutants from the previous analysis. The following steps consisted of prompt definition and execution. During this step, the maximum number of interactions with the LLM was increased from three to five to see whether it would produce different results. Then, the generated test was executed against the set of live mutants, and the score was computed and validated using statistical testing.

6.6.1 Prompt – 2nd Experimental Study

The prompt structure used in this experiment is similar to that of the previous one, as shown in Figure 68.

The major changes of the second prompt were that the mutated method and a brief description of the mutant were provided. The latter change aims to help the LLM better understand context. For instance: *The mutation operator mutates the current instantiation of "BluetoothAdapter adapter = BluetoothAdapter.getDefaultAdapter();" with null. That is, "BluetoothAdapter adapter = null".* In case of compilation/execution error, the same prompt presented in Figure 50 was adopted. Additionally, the LLM provided the following information whenever the test ran successfully in both the original and mutated code: *The test works; however, the original program and the mutant produce the same output.* Deeply analyze the class under testing and provide additional assertions to ensure the test passes in the original program and fails in the mutant. Just provide the test case without any additional comments.

For the second experimental study, we used only Claude as the LLM because it performed slightly better than the others. Additionally, we ran the prompts in OpenRouter.

```
Your role is a senior Android tester with broad experience in mutation
testing. Your task is to provide an Android instrumented test case to kill a
specific mutant. Following, I provide the ***ORIGINAL CLASS*** under test --
<FULLY_QUALIFIED_CLASS_NAME>:

***ORIGINAL CLASS***
-----------------------------------------------------------------
<CLASS_UNDER_TEST>
-----------------------------------------------------------------
***MUTATED METHOD***
<MUTATED_METHOD>
-----------------------------------------------------

Analyze the ***ORIGINAL CLASS*** and the ***MUTATED METHOD*** above.

<DESCRIPTION_OF_MUTANT>

Create a ***single test case*** to kill this mutant.

Constraints:
- A single Android instrumented test case in JUnit 4 format and AndroidX
package.
- For private mutated methods, you need to access class state via public
method to show the different behaviour from the ***ORIGINAL CLASS*** to the
***MUTATED METHOD***
- Do not use mock.
- Do not provide additional comments or explanations.
- The test case must call: <TEST_CLASS_NAME>
- The test package is: <TEST_PACKAGE>
```

Figure 68 – Prompt for mutation-guided test case generation.

6.6.2 Results

Table 58 presents the results of the second experimental study. Column $LIVE_C$ refers to the number of mutants that remained alive for the Claude test set. Column N_{TEST} presents the number of executable tests generated by Claude. Column N_{KILLED} displays the number of killed mutants, and Column MS_{C2} provides the mutation score.

The approach killed 12 out of 64 live mutants using 10 test cases. This shows that some of these could kill more than one mutant (e.g., AOBD and RU). Consequently, the mutation score of some applications had also increased, as presented in Figure 69.

In seven out of ten applications, the mutation score increased (AOBD, ASCL, AUTH, C2T, GB, RU, and WS), whereas in three of them the score remained the same (A2DP, GPSL, and ULOG). The average mutation score increased from 41.18% (see Table 52) to

Table 58 – Results of mutation-guided test case generation.

App	$LIVE_C$	N_{TEST}	N_{KILLED}	MS_{C2}
A2DP	4	0	0	20.00%
AOBD	5	1	2	57.14%
ASCL	19	1	1	14.29%
AUTH	2	1	1	87.50%
C2T	6	3	3	78.57%
GB	7	1	1	53.85%
GPSL	5	0	0	28.57%
RU	13	2	3	44.44%
ULOG	2	0	0	33.33%
WS	1	1	1	100.00%

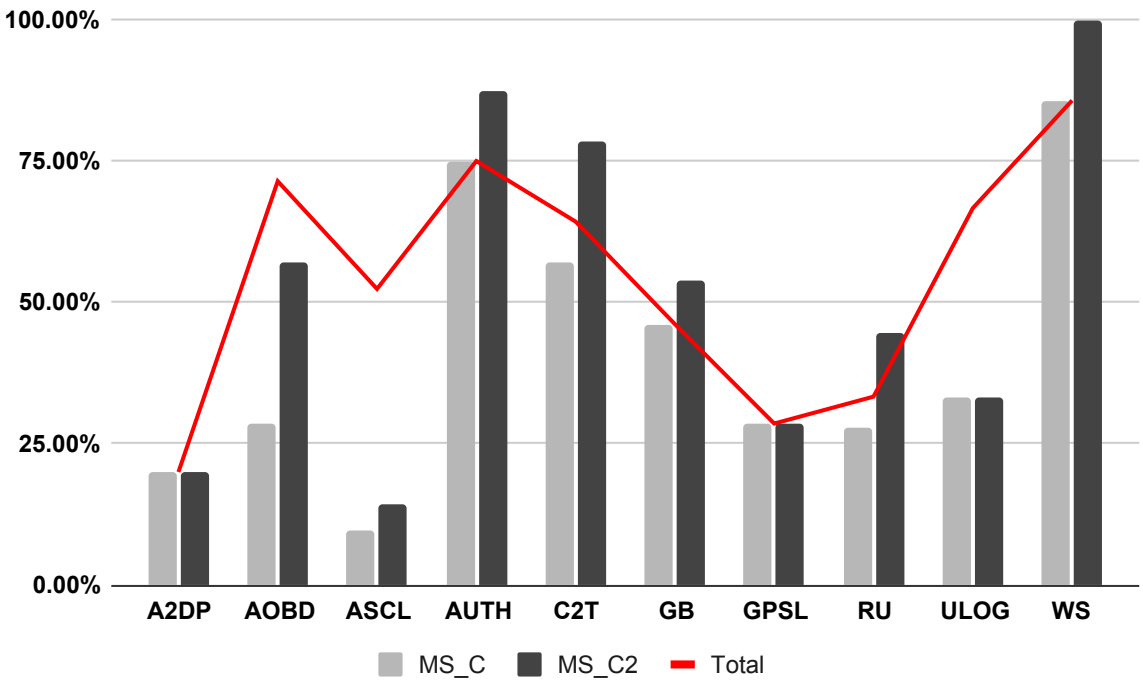


Figure 69 – Comparing MS_C to MS_{C2}.

51.77%. Comparing the "Total" mutation score, the second approach increased it in five of ten applications. In two out of ten apps, the score remained the same. And in three of them, the combination of all test cases outperformed the current approach.

Next, we statistically evaluated the results. First, we ran Shapiro-Wilk tests to verify whether the sample came from a normal distribution at the 95% confidence level. In all three cases, the p-values were higher than 0.05; hence, we assumed that the data were normally distributed.

Therefore, the validation used the Student's t-test to assess whether there was a statistical difference among the samples at the 95% confidence level. For this analysis, the following hypotheses were defined:

- **Null Hypothesis (H_0):** There is no difference among the mutation scores of MS_C and MS_{C2}
- **Alternative Hypothesis (H_1):** There is a difference among the mutation scores of MS_C and MS_{C2}.

Since the p-value is greater than 0.05, we rejected the alternative hypothesis, indicating that there was no statistical difference in mutation scores.

Next, we performed the Cohen's d effect size to validate whether the MS_{C2} was larger than MS_C. After running this test, we obtained a small-to-medium effect size of 0.3924. Therefore, we can infer that MS_{C2} is slightly larger than MS_C.

6.6.3 Prompt Improvement – New Attempt

Since the results of the current experimental study seem promising, we decided to improve our prompt and seek to kill the remaining live mutants.

In this phase, we assessed the live mutants and observed that the LLM struggled to generate tests for private methods and to handle exceptions. Therefore, the prompt was augmented with guidelines to treat these issues.

For private methods, we develop a tool that generates a dependency tree showing which methods directly or indirectly call the methods of an application. We found that this information could help the LLM determine which public methods to consider when generating tests for private methods. Additionally, we detailed in the prompt that exceptions may occur in the mutated code rather than in the original. Therefore, Figure 70 presents a template of this augmented prompt.

Moreover, Figure 71 exemplifies an excerpt of the dependency tree. It shows the class and its implemented methods, along with their access modifiers. A method can be "External", i.e., is implemented in a different class, whereas "Internal" is a method from the current class.

Table 59 summarizes the results of the second attempt. Column $LIVE_{C2}$ presents the number of live mutants of the previous analysis. Columns N_{TEST}, N_{KILLED}, and MS_{C3} present the number of tests, number of killed mutants, and final mutation score using the augmented prompt.

As shown, Claude generated tests to kill mutants across three applications: ASCL, C2T, and GPSL. Considering the first app, the LLM was able to kill more mutants (6)

```
Your role is a senior Android tester with broad experience in mutation testing. Your task is to provide
an Android instrumented test case to kill a specific mutant. Following, I provide the ***ORIGINAL
CLASS*** under test -- <FULLY_QUALIFIED_CLASS_NAME>:
-----------------------------------------------
***ORIGINAL CLASS***
<CLASS_UNDER_TEST>
-----------------------------------------------
***MUTATED METHOD***
<MUTATED_METHOD>
-----------------------------------------------
Deeply analyze the ***ORIGINAL CLASS*** and the ***MUTATED METHOD*** above.
Do create a single test case to kill this mutant.

CONSTRAINTS:
A single Android instrumented test case in JUnit 4 format and AndroidX package.
For private mutated methods, you need to access the class state via a public method to show the different
behaviour from the ***ORIGINAL CLASS*** to the ***MUTATED METHOD***
Do not use mock.
Do not provide additional comments or explanations.
The test case must call: <TEST_CLASS_NAME>
The test package is: <TEST_PACKAGE>
Create a test that calls the mutated method indirectly through a public method, considering the
DEPENDENCY TREE below.

-----------------------------------------------
***DEPENDENCY TREE***
<DEPENDECY_TREE_CODE>
-----------------------------------------------
***ADDITIONAL CONSTRAINTS***:
The test must:
PASS (run successfully) on the ***ORIGINAL CLASS***
FAIL on the ***MUTATED METHOD***
This is for mutation testing - we need to "kill the mutant" by creating a test that demonstrates
different behavior between original and mutated code.

The test must demonstrate different behavior between the original and mutated code, WITHOUT expecting
exceptions. The original code should run normally and produce some observable behavior, while the mutated
code should produce different observable behavior (which may include exceptions). The test should assert
on the normal behavior expected from the original code, and this assertion should fail when run against
the mutated code.
```

Figure 70 – Improved prompt for mutation-guided test case generation.

```
--- File: GpsStatus.java ---
    Caller: public  GpsStatus.onStatusChanged(String, int, Bundle)
        Callee: provider.equalsIgnoreCase (External)
        Callee: clear (Internal)
        Callee: listener.onTick (External)
    Caller: public  gpsStatusListener.onGpsStatusChanged(int)
        Callee: locationManager.getGpsStatus (External)
        Callee: gpsStatus.getSatellites (External)
        Callee: satellite.usedInFix (External)
        Callee: listener.onTick (External)
    Caller: private  GpsStatus.clear(boolean)
    Caller: public  GpsStatus.isLogging()
    Caller: public  GpsStatus.isFixed()
    Caller: public  GpsStatus.getSatellitesAvailable()
    Caller: public  GpsStatus.getSatellitesFixed()
    Caller: public  GpsStatus.isEnabled()
        Callee: context.getSystemService (External)
        Callee: Objects.requireNonNull(lm).isProviderEnabled (External)
        Callee: Objects.requireNonNull (External)
```

Figure 71 – Code snippet of the dependency tree.

Table 59 – Results of mutation guided test case generation – Second attempt.

App	$LIVE_{C2}$	N_{TEST}	N_{KILLED}	MS_{C3}
A2DP	4	0	0	20.00%
AOBD	3	0	0	57.14%
ASCL	18	6	7	47.62%
AUTH	1	0	0	87.50%
C2T	3	1	1	85.71%
GB	6	0	0	53.85%
GPSL	5	1	2	57.14%
RU	10	0	0	44.44%
ULOG	2	0	0	33.33%
WS	0	0	0	100.00%

compared to the previous phases: 2 in the first attempt and 1 in the second. In the case of GPSL, the LLM could generate tests to mutate mutants that did not occur in the previous phase. Figure 72 summarizes the comparison between MS_{C2} and MS_{C3}.

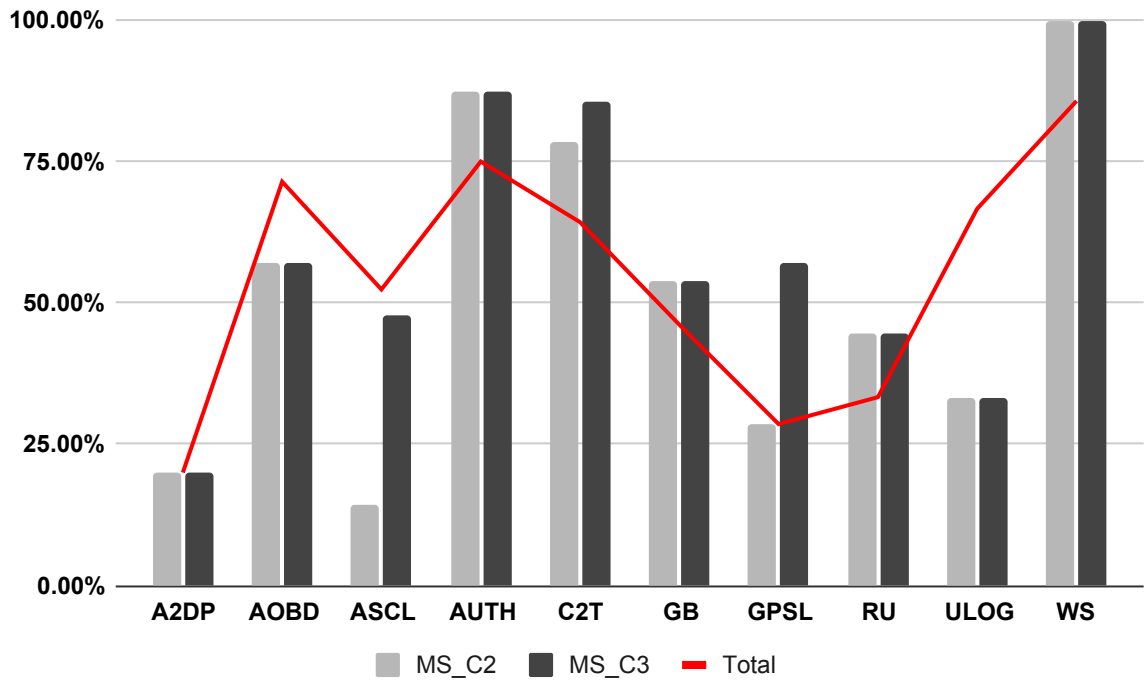


Figure 72 – Comparing the MS_{C2} and MS_{C3}.

It is possible to observe a significant difference between the two scores for the apps ASCL and GPSL. However, for these two applications and ULOG, combining test sets from other LLMs still yielded better scores.

We performed statistical tests to conclude the analysis by comparing MS_C and MS_{C3}. In other words, we aimed to evaluate whether there is any statistical difference in adopting a mutation-guided test case generation approach. Thus, the following hypotheses were defined:

- **Null Hypothesis (H_0):** There is no difference among the mutation scores of MS_C and MS_{C3}.
- **Alternative Hypothesis (H_1):** There is a difference among the mutation scores of MS_C and MS_{C3}.

Since MS_{C3} presents a normal distribution, we used the Student's t-test to compare the sample at the 95% confidence level. Once the resulting p-value was 0.1338, we rejected the alternative hypothesis, showing that there is no difference between the two mutation scores.

Finally, we computed Cohen's d effect size, which yielded a medium-to-large effect size of 0.7021. Using this metric, we argue that the mutation-guided test case generation approach achieved a higher mutation score than the initial approach.

6.7 Qualitative Analysis – Mutation Operators

This section provides a brief qualitative analysis of the mutation operators. Recapping Table 51, six out of ten mutation operators generated mutants for the subject apps. NDID generated 76 mutants, followed by DDM (10), TDC (6), RLPDR (5), RAIDD (4), and BDL (2). NDID was the only mutation operator that generated mutants for all applications.

Now, we investigate the characteristics of the mutation operators for the mutants that were killed and those that remained alive during the three test case generation phases presented earlier. Based on this information, we aim to provide some basis for prompt enhancement. That is, if we can identify some characteristics of the mutation operators to facilitate the mutation-guided test case generation strategy.

Initially, Table 60 presents the statistics regarding the mutation operators of the killed mutants of each LLM. Column *Total* represents the number of mutants per operator. K_C, K_G, K_D, and K_Q represent the number of killed mutants of Claude, ChatGPT, DeepSeek, and Qwen, respectively. Finally, K_T is the number of killed mutants combining all four test sets.

Considering Claude, 35 of the killed mutants were generated via NDID, two via BDL, one via RLPDR, and one via TDC. The test set killed 46.05% of NDID mutants and 100% of BDL mutants. On the other hand, it could not kill any mutant from DDM and RAIDD.

ChatGPT and DeepSeek killed the mutants using the same type of mutation operators. ChatGPT killed 24 mutants from NDID and one from RLPDR, whereas DeepSeek killed 26 from NDID and one from RLPDR. Moreover, Qwen killed 30 mutants generated by four mutation operators: NDID, BDL, RAIDD, and TDC.

Table 60 – Number of mutants killed per mutation operator.

Operator	**Total**	$\mathbf{K_C}$	$\mathbf{K_G}$	$\mathbf{K_D}$	$\mathbf{K_Q}$	$\mathbf{K_T}$
BDC	0	0	0	0	0	0
BDL	2	2	0	0	1	2
DDBID	0	0	0	0	0	0
DDM	10	0	0	0	0	0
NDID	76	35	26	24	27	50
RAIDD	4	0	0	0	1	1
RLPDR	5	1	1	1	0	1
RLRBPDR	0	0	0	0	0	0
TDC	6	1	0	0	1	2
TDL	0	0	0	0	0	0

Considering the total, 65% of mutants generated through NDID were killed by the combined test sets. As seen before, all mutants from BDL were killed by the test sets. Thus, we can infer that the test sets of the first round are more likely to kill mutants from these two types of mutation operators. On the other hand, the LLMs could not generate tests to kill mutants of the DDM operator.

Next, we evaluated the two mutation-guided test case generation approaches. Figure 73 illustrates the results.

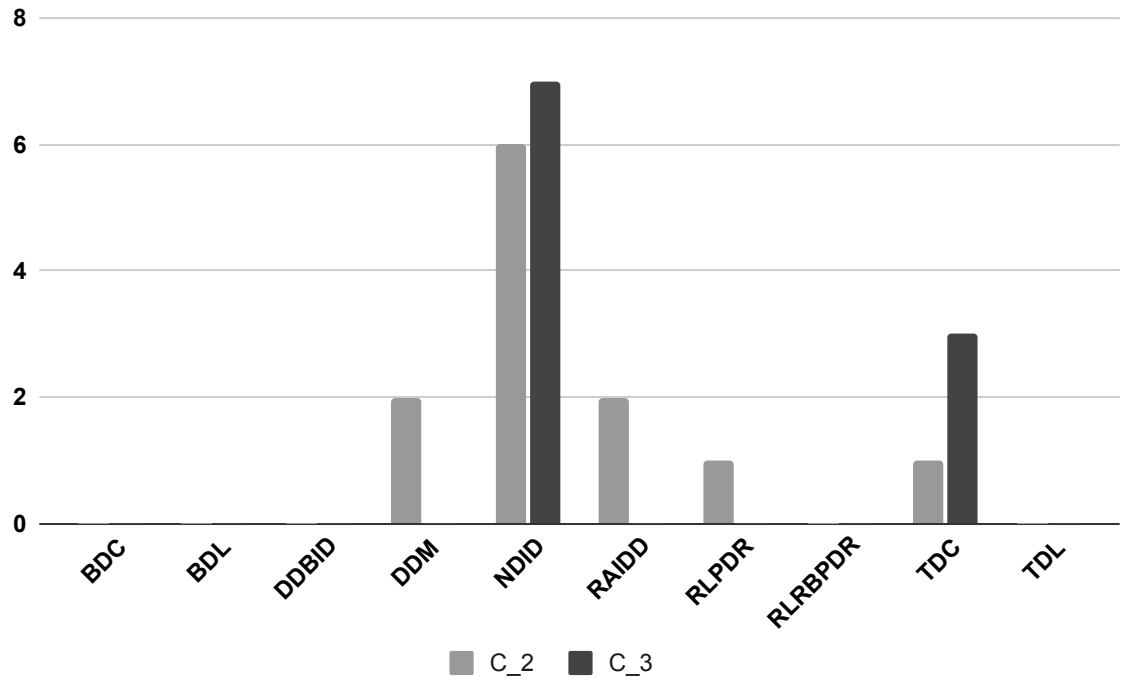


Figure 73 – Comparing killed mutants from both mutation-guided attempts.

Once again, the test set from Claude, considering a more specific prompt, killed more mutants from NDID. However, it killed mutants from RAIDD, RLPDR, and TDC. Two

mutants from DDM were killed in the first round of the mutation-guided test case generation. Furthermore, the second attempt of the test generation strategy managed to kill almost all of the mutants generated by TDC, leaving only one mutant alive.

To summarize, after the three rounds of test generations, the number of killed mutants per mutation operator is:

- BDL: 2 (100%).
- DDM: 2 (20%).
- NDID: 48 (63.16%)
- RAIDD: 2 (50%)
- RLPDR: 2 (40%)
- TDC: 5 (83.33%)

6.8 Discussion of the Results

This paper investigates the capabilities of four large language models in generating unit test cases for two Android device components: Bluetooth and Location APIs. To this aim, we adopted two strategies for test case generation: (i) starting from scratch and (ii) mutation-guided test case generation. In the first strategy, we assumed that the applications had no tests or, when they did, they did not exercise the parts we needed (i.e., the classes that implemented Bluetooth or Location functionalities). In fact, this issue reinforces the findings of Pecorelli et al. (2021), which show that few Android open-source applications had tests. In our case, we found that only two applications had instrumented tests implemented (i.e., ASCL and AUTH). In both cases, the mutation score for the tests from these two apps was 0.

The latter strategy focused on providing test cases based on a specialized prompt, using information from live mutants from the previous stage. The tests were evaluated based on code coverage, mutation score, and the chrF similarity metric.

For the first strategy, we generated a single test set for each LLM and app. Considering overall coverage, Claude (59.36%) has the best average performance, followed by ChatGPT (54.18%), DeepSeek (42.02%), and Qwen (38.45%). Statistically, we observed no significant difference in average code coverage between Claude and ChatGPT. However, the number of executed tests provided by Claude was three times larger than ChatGPT. Therefore, we can infer that ChatGPT produces more useful tests than Claude. Moreover, pairing the test sets indicates a complementarity between them. It shows that a test set generated by a given LLM may cover different scenarios not captured by others.

Next, we evaluated the quality of each test set based on its fault-detection capabilities. Initially, we generated mutants considering ten mutation operators. Overall, the number of mutants for the subject apps may be considered low compared to traditional mutation operators; however, since we are dealing with a specific type of mutants, it is somewhat expected.

Once again, Claude (41.18%) outperformed DeepSeek (30.25%), ChatGPT (27.72%), and Qwen (28.09%) in terms of average mutation score. Although there was no significant difference between the test sets, Cohen's d effect size shows that Claude has a superior mutation score compared to the other samples. Additionally, we computed the time spent running mutation testing considering the generated test set. Despite focusing on unit tests, which tend to be faster than GUI testing, we spent approximately 7 days and 20 hours running the first phase of the experimental study.

In an ideal scenario, we would create more test sets for each LLM to minimize potential bias and compute the average metric for each test set. Using this strategy, we would spend at least 70 days (or two months and ten days) to run the first phase of the experiment. Note that we conducted two complementary test case generation stages, which can increase the total experimental time. One can suggest parallelizing the tasks, i.e., generating the test sets while executing mutation testing. However, it would be infeasible since we use a single mobile device. Hence, we decided to make a single run given the time constraint.

Then, we evaluated the mutation score by pairing the test set. In general, the average mutation score increased for the paired combinations. However, Table 55 shows that there is little complementarity among the samples. In fact, the gain degree may be inflated by the AOBD app, which had the largest mutation score among the pairing combinations. Moreover, DeepSeek had a larger complementarity impact.

We ran a Pearson Correlation test between code coverage and mutation score. The results show an r-value of 0.013 and a p-value of 0.970, indicating no correlation between code coverage and mutation score across all test sets generated by the LLMs.

To conclude the analysis considering the first strategy, we compute the chrF similarity metric to verify the similarity among the test sets generated by the LLMs. The overall results showed a low similarity between the samples. In fact, none of the samples achieved a similarity larger than 30%. On the other hand, we found some outliers. For instance, we had one case with 100% similarity. In this case, Claude and DeepSeek generated the same test cases. Despite that, we could not make any correlation between chrF and the other metrics.

In the second approach, we assumed a test set existed and adopted a strategy to improve it by generating test cases from live mutants. For this analysis, we chose Claude because it achieved the best results for code coverage and mutation score.

For this second experimental study, we devised a more structured prompt that provided information about the mutant to be killed. Additionally, we requested only a single test case rather than a test set. The overall results showed that we could kill 18% of live mutants. For the WS app, the mutation score reached 100%, i.e., all mutants were killed.

Next, we improved the prompt by including the class under test's dependency tree. In this phase, we killed ten out of 52 live mutants, of which seven were from the AOBD app.

After analyzing the remaining live mutants, we observed that most were related to a Service and/or Broadcast Receiver component. In this case, the tests' generation or the infrastructure on which they are executed may not be sufficient to run them. For instance, depending on the application, we may require a Bluetooth device to trigger communication between the app and the device to reliably validate a given functionality during testing, which may increase the cost and complexity of the test. One may suggest using mocks for validation. However, it may not mimic the real behavior of the tests and the application.

Finally, we carried out a qualitative analysis of the mutation operators. The results indicate that mutants generated from NDID and BDL are easier to kill than those generated from the others. For instance, during the first phase of the experimental study, we were unable to kill any mutants generated by DDM.

After extending the experimental study, we were able to kill mutants from another class. For the TDC mutation operator, the tests killed five of six mutants generated by it.

For future work, we intend to explore using a more specialized prompt tailored to the type of mutation operator. That is, we would provide a specific prompt for each class of mutation operator, tailored to its characteristics, to kill the remaining live mutants.

6.9 Related Works

Recently, the number of studies using LLMs to generate test cases has grown considerably. In this paper, we pinpoint some of these studies.

Yi et al. (2023) investigates the capability of ChatGPT in generating Java unit test cases, using model GPT-3.5-turbo. The authors compared the tests against EvoSuite (FRASER; ARCURI, 2011) and manual testing. The authors showed that ChatGPT performed poorly compared to the other approaches. (OUEDRAOGO et al., 2024) assessed the effectiveness of four LLMs (GPT3.5, GPT-4, Mistral 7B, and Mixtral 8x7B) in generating unit test cases and the impacts of five prompt engineering techniques: zero-shot learning, few-shot learning, chain-of-thought, tree-of-thoughts, and guided tree-of-thoughts. Based on the results, the authors found that EvoSuite remains the baseline for code coverage and bug detection.

Siddiq et al. (2024) investigates the capabilities of three LLMS (i.e., Codex, GPT-3.5-Turbo, and StarCoder to generate unit tests for two datasets: HumanEval and SF110. The generated tests were evaluated based on compilation status, correctness, coverage, and test smells. Once again, EvoSuite showed slightly better results, outperforming LLMs. Yang et al. (2024) investigated the effectiveness of CodeLlama-7B-Instruct, CodeLlama-13B-Instruct, Phind-CodeLlama-34B-v2, DeepSeekCoder-6.7B-Instruct, and DeepSeekCoder-33B-Instruct, and compared to GPT-4 and EvoSuite. In this case, the tests were evaluated based on compilation success rate, line and branch coverage, and the number of detected defects.

Chen et al. (2025b) presented an approach for unit test generation based on a three-module prompt. The tests were compared against EvoSuite, AthenaTest, and ChatTester according to test correctness and coverage. The approach outperformed ChatTester and AthenaTester in terms of correctness. Considering coverage, the approach outperformed EvoSuite in line coverage but failed in branch coverage. Nan et al. (2025) presents IntUT, an approach that leverages test intentions such as input, mocks, and expected outputs, aiming to increase focal methods' branch coverage.

In Android, most approaches focus on generating test cases or test scripts to validate the GUI functionality of an application under test (CHEN et al., 2025a). Liu et al. (2023b) proposed QTypist, an approach that leverages an LLM for generating input text. Overall, the proposed technique achieves a superior GUI page-passing rate compared to the baseline approaches.

García et al. (2024) assessed whether the use of ChatGPT can optimize the automated generation of end-to-end tests for Android by serving as an assistant to reduce the effort of generating these types of tests. Liu et al. (2024) designed GPTDroid, a technique that models the prompt using a Q&A approach that leverages GUI context information. The results showed that the approach outperformed the baseline regarding activity coverage and bug detection.

Phung et al. (2025) proposed VisiDroid, an approach that generates test cases and scripts from natural-language descriptions of Android tasks, using screenshots as the oracle and AI agents as the script generator. The results indicated that the approach outperformed the baseline methods regarding completion rate and action accuracy. Wang et al. (2024) provides an approach that combines static analysis information, the CO-STAR framework for prompt designing, and LLM for test case generation. Kong et al. (2025) devised ProphetAgent, an approach to synthesize executable GUI testing from test cases written in natural language and GUI transitions information. The overall results showed that the approach achieved an execution rate of over 78% and optimized testing by reducing effort.

(WANG et al., 2025) designed LLMDroid, a framework to improve the performance of existing GUI testing tools by using code coverage as a metric to optimize test generation.

The approach generally uses existing tools to explore the existing app pages. Then the LLM works to improve the unexplored pages and consequently increase code coverage. The results showed an average increase of 26% for code coverage and 29.31% for activity coverage. (YOON et al., 2025) presented an approach that resembles the QTypist approach, that is, it aims to generate text input for apps using contextual information and a more structured prompt.

We could not find studies on generating Android instrumented unit tests for device components such as Bluetooth and Location. Therefore, we argue that our work makes a novel contribution to the field of research.

6.10 Threats to the Validity

Construct Validity

Considering construct validity, the samples were measured in terms of code coverage. Since the studies focus on test generation for Bluetooth and Location components, only the classes that directly or indirectly implement their functionalities are examined. Therefore, the unrelated classes were excluded to avoid skewing the data.

Regarding mutation testing, all apps used the same mutation operators and the same parameters for a fair comparison. Additionally, the implementation of mutation operators may be a threat. The mutation operators were implemented within an existing framework (i.e., METFORD (VINCENZI et al., 2025)) and followed the same programming design as the existing mutation operators.

Finally, the last threat consists of the LLMs used in this paper. We tried to use LLMs available from different providers that had been adopted in previous research.

Internal Validity

Regarding internal validity, we standardized all data-collection resources used in the experimental study. For instance, we utilized the same prompt in all LLMs. Using different prompts may produce different results.

All subject apps were built using the same Java version and the same build system (i.e., Gradle). Moreover, we used the same environment configuration to run the generated test cases. We started a clean test execution between the LLMs to avoid any previous state affecting the current execution.

External Validity

The results of this study may not be generalizable. We adopted a set of ten real, publicly available apps. We followed a well-defined set of selection criteria to minimize potential threats.

Additionally, we implemented a subset of the mutation operators from Kuroishi et al. (2025). Therefore, using different mutation operators and different applications may produce different results.

Conclusion Validity

The main threat to conclusion validity is using a single test set for validation. Ideally, we would generate multiple test sets for each LLM on different days to minimize potential threats (SALLOU; DURIEUX; PANICHELLA, 2024). However, given the time constraints, we decided to utilize a single test set.

Moreover, we repeated the experimental study five times to avoid the effects of flaky tests Thorve, Sreshtha e Meng (2018). Given the non-deterministic nature of this test type, we skipped its execution to avoid skewed data.

Finally, we adopted different statistical tests to validate the results. However, the limited sample size (i.e., 10) may skew the reliability of the statistical results. Additionally, the statistical tests used to assess group differences and the effect size were based on Shapiro-Wilk test results, which may have influenced the selection of tests. Using a larger sample may yield a different analysis, indicating that the sample does not follow a normal distribution, which is usually characteristic of code coverage and mutation score.

6.11 Conclusion and Future Works

Although testing activities are fundamental in the software development cycle, several open-source applications still lack tests to validate their functionality (PECORELLI et al., 2021). The complexity of testing an Android app compared to web and desktop applications may explain the lack of test cases for this type of app (MUCCINI; FRANCESCO; ESPOSITO, 2012).

Recently, many researchers have started investigating the adoption of GenAI to solve different Software Engineering problems. Specifically, we observed an increase in studies exploring LLMs for test case generation. Considering the Android ecosystem, most studies focus on generating GUI tests given the event-based nature of these apps (AMALFITANO et al., 2013; AMALFITANO et al., 2015).

On the other hand, few studies focus on generating test cases that consider different Android device components, such as location, connectivity, and sensors. Given this research gap, we proposed an initial investigation into test case generation that considers

location and Bluetooth Android components. To this end, we leveraged an LLM to generate test cases for these hardware components and evaluated them using code coverage, mutation testing, and the chrF similarity metric.

This paper adopted two strategies for test case generation. The first involved starting from scratch, assuming that the apps did not have any implemented. The second involved a mutation-guided test case generation strategy focused on generating tests to kill specific mutants.

Considering the first one, Claude has the best code coverage and mutation score performance, though it does not show a significant statistical difference compared to the other LLMs. ChatGPT had an interesting performance in code coverage, but the worst in mutation score. Moreover, Qwen had the highest code coverage complementarity, whereas DeepSeek had the best mutation score complementarity.

In the second strategy, we selected Claude as the baseline since it performed slightly better than the other LLMs. In this case, we created a specific prompt to generate test cases that kill mutants that survived the first stage. We divided this analysis into two parts, and the main difference between them concerns the information about the application under test's dependence tree.

Overall, we increased the mutation score by killing an additional 22 out of 64 mutants. In three apps, we achieve a mutation score over 85%, and in one of them, the tests kill all mutants. The main advantage of the second strategy is that we can restrict the number of tests by setting a goal based on the type of defects the test must detect.

Among the mutants that remained alive, many are related to the Service or BroadcastReceiver components of the Android API. In this case, we are unaware whether the traditional unit tests can kill these mutants. Depending on the type of mutants, we advocate considering additional test aspects, such as specialized testing infrastructure or a testbed, to emulate real conditions and achieve more reliable validation.

In summary, the results of this work appear promising and provide a basis for testing different components of Android devices. This led us to understand the advantages and limitations of the strategies used in this work, which, in addition to providing a broad view of future works:

- ❏ Perform a financial cost analysis in relation to the cost per number of tests generated.
- ❏ Extending the experimental study aiming at GUI testing of Bluetooth and Location components.
- ❏ Extending the experimental study to other Android device components (e.g., Wi-Fi and sensors).

- ❏ Evaluate different LLMs for the mutation-guided test case generation approach. We observed that DeepSeek had the best complementarity gain among the LLMs. Therefore, we intend to verify the LLM's ability to enhance an existing test set.
- ❏ Apply different quality metrics to the generated test sets, for instance, test smell rate and sustainability-related metrics.

Chapter 7

Conclusion and Future Works

7.1 Conclusion

From simple desktop applications to the use of large language models to solve complex tasks, the technological advances and innovations that emerge daily offer opportunities to advance an area of knowledge and make diverse contributions from both academic and industrial perspectives. In this thesis, we sought to make several contributions to improve the existing mobile application testing process, considering both theoretical and practical aspects. Since this thesis presents four published papers and one in submission, we are confident that we have made solid contributions to the community.

To guide the PhD project, we were inspired by the experience report of Kudo et al. (2022) and adopted a systematic approach to guide our work. In general, this approach helped us gain a broad overview of the work to be done and facilitated the definition of checkpoints to be conquered along the way.

Regarding the contributions, the first paper published during the PhD was a tertiary study in which we analyzed 21 secondary studies on mobile application testing. This work used a systematic procedure (KITCHENHAM; CHARTERS, 2007) to minimize possible bias. The paper was designed to answer three main questions: we provided a characterization of the studies based on the frequency, types, and quality of secondary studies; the research topic, test objectives, and test platforms; and an in-depth analysis of the main research gaps identified by each secondary study. At the end, we defined a research agenda with 15 open challenges, some of which served as a basis for further work.

The second paper provides a fine-grained mapping study that characterizes existing testing infrastructures to support mobile application testing. In the paper, we contributed to classifying 27 primary studies that proposed different testing infrastructures based on

testing type, supported devices, and operating systems. We also evaluated the availability of each infrastructure and the testing types and applications it supports. Moreover, we also provided a new classification based on the mapped studies.

The rationale for a mapping study focusing on testing infrastructures was due to the academia-industry partnership that sponsored this PhD, as detailed in Chapter 1. This provided an overview of the literature and guided us in developing the third paper of this thesis, which compared existing testing infrastructures and defined a local device farm to be implemented in the company, along with a CI/CD pipeline.

Once the infrastructure was defined, we focused on enhancing the testing process by investigating mutation testing for Android device components and automated test case generation. The fourth paper of this thesis proposes a set of 16 mutation operators focusing on Bluetooth, AltBeacon, and Location components of the Android environment. The rationale for considering these components was the type of application developed by the partner company (see Chapter 1 for more details). This led us to derive a set of 16 mutation operators categorized as: Replacement, Null, Swift, Trap, Deletion, and Shift. Additionally, we extended the set of previously implemented operators in MET-FORD (VINCENZI et al., 2025), which we validated during the PhD "sandwich".

Finally, in the last paper, currently under submission, we investigated the capabilities of existing LLMs for automatic test case generation for Bluetooth and Location components. In this paper, we provided an extensive work using 10 open-source applications. We found that LLMs can generate practical tests but still lack robustness when depending on the API structure (e.g., Service and BroadcastReceiver).

Despite the effort and contributions of this thesis, we identify research gaps for future work in the next section.

7.2 Future Works

- ❏ Integrate the practical contributions into a single solution, providing a testing infrastructure, or a local device farm, capable of generating, executing, and validating the test cases, considering Bluetooth and Location apps in a CI/CD scenario.
- ❏ Extending the proposed mutation operators for other Android components.
- ❏ Investigate automated test case generation considering the same components but with a view to the GUI.
- ❏ Investigate other metrics for validating the test generated by the LLM, such as test smells.

- ❏ Investigate different aspects of using LLM in the testing process. We observed that many studies focus on technical aspects, but few consider the sustainable, ethical, and human aspects of LLM use.
- ❏ Definition of an incremental generation strategy by combining different LLMs to improve test set quality.
- ❏ Define a mutation testing strategy based on specific mutation operators aiming at cost reduction, keeping almost the same efficacy of the generated test set.
- ❏ Contribute to the development of a benchmark for evaluating testing criteria on mobile applications.

Bibliography

ABDI, M.; DEMEYER, S. Steps towards zero-touch mutation testing in Pharo. In: **21st Belgium-Netherlands Software Evolution Workshop – BENEVOL'2022**. Mons: [s.n.], 2022. (CEUR Workshop Proceedings, v. 1), p. 10.

AGRAWAL, H. et al. **Design of mutant operators for the C programming language**. [S.l.], 1989.

AL-AHMAD, A. S.; ALJUNID, S. A.; ISMAIL, N. K. Mobile cloud computing applications penetration testing model design. **Int. J. Inf. Comput. Secur.**, Inderscience Publishers, Geneva 15, CHE, v. 13, n. 2, p. 210–226, jan 2020. ISSN 1744-1765.

AL-AHMAD, A. S.; ALJUNID, S. A.; SANI, A. S. A. Mobile cloud computing testing review. In: **2013 International Conference on Advanced Computer Science Applications and Technologies**. [S.l.: s.n.], 2013. p. 176–180.

AL-AHMAD, A. S. et al. Systematic literature review on penetration testing for mobile cloud computing applications. **IEEE Access**, v. 7, p. 173524–173540, 2019.

ALI, A.; MAGHAWRY, H. A.; BADR, N. Automated parallel gui testing as a service for mobile applications. **Journal of Software: Evolution and Process**, v. 30, n. 10, p. e1963, 2018. E1963 JSME-17-0149.R3.

ALMEIDA, D. R.; MACHADO, P. D. L.; ANDRADE, W. L. Testing tools for android context-aware applications: a systematic mapping. **Journal of the Brazilian Computer Society**, v. 25, n. 1, p. 12, Dec 2019. ISSN 1678-4804.

ALMEIDA, D. R. de; MACHADO, P. D. L.; ANDRADE, W. L. Context-aware android applications testing. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (SBES '20), p. 283–292. ISBN 9781450387538.

AMALFITANO, D. et al. Using gui ripping for automated testing of android applications. In: **2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering**. [S.l.: s.n.], 2012. p. 258–261.

AMALFITANO, D. et al. Considering context events in event-based testing of mobile applications. In: **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops**. [S.l.: s.n.], 2013. p. 126–133.

AMALFITANO, D. et al. Mobiguitar: Automated model-based testing of mobile apps. **IEEE Software**, v. 32, n. 5, p. 53–59, 2015.

AMALFITANO, D. et al. How do java mutation tools differ? **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 65, n. 12, p. 74–89, nov 2022. ISSN 0001-0782.

AMANO, T. et al. Smartphone applications testbed using virtual reality. In: **Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services**. New York, NY, USA: Association for Computing Machinery, 2018. (MobiQuitous '18), p. 422–431. ISBN 9781450360937.

AMAZON. **AWS Device Farm**. 2023. <https://aws.amazon.com/device-farm>. Last accessed: February 2023.

AMAZON. **AWS Device Farm**. 2024. <https://aws.amazon.com/device-farm>. Last accessed: February 2024.

Android Developers. **Fundamentals of testing Android apps**. 2023. <https://developer.android.com/training/testing/instrumented\protect\discretionary{\char\hyphenchar\font}{}{}tests>. Last accessed: February 2023.

ARAUJO, R. F. et al. Devising mutant operators for dynamic systems models by applying the hazop study. In: **International Conference on Software Engineering Advances - ICSEA 2011**. Barcelona, Spain: IARIA - The International Academy Research and Industry Association, 2011. p. 58–64.

ARIF, K. S.; ALI, U. Mobile application testing tools and their challenges: A comparative study. In: **2019 2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)**. [S.l.: s.n.], 2019. p. 1–6.

ASCATE, S. M. et al. Challenges in model-based testing for mobile applications. In: **Proceedings of the XX Iberoamerican Conference on Software Engineering, Buenos Aires, Argentina, May 22-23, 2017**. [S.l.]: Curran Associates, 2017. p. 567–580.

AZIM, T.; NEAMTIU, I. Targeted and depth-first exploration for systematic testing of android apps. In: **Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications**. New York, NY, USA: Association for Computing Machinery, 2013. (OOPSLA '13), p. 641–660. ISBN 9781450323741.

BETKA, M.; WAGNER, S. Towards practical application of mutation testing in industry – Traditional versus extreme mutation testing. **Journal of Software: Evolution and Process**, v. 34, n. 11, p. e2450, 2022.

BINH, N. T. et al. Experience report on developing a crowdsourcing test platform for mobile applications. In: HERNES, M.; WOJTKIEWICZ, K.; SZCZERBICKI, E. (Ed.). **Advances in Computational Collective Intelligence**. Cham: Springer International Publishing, 2020. p. 651–661. ISBN 978-3-030-63119-2.

BOEHM, B.; BASILI, V. R. Software defect reduction top 10 list. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 34, n. 1, p. 135–137, jan. 2001. ISSN 0018-9162.

BRADBURY, J. S.; CORDY, J. R.; DINGEL, J. Mutation operators for concurrent java (j2se 5.0). In: **Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)**. Raleigh, NC, USA: IEEE, 2006. p. 11–11.

BRERETON, P. et al. Lessons from applying the systematic literature review process within the software engineering domain. **Journal of Systems and Software**, v. 80, n. 4, p. 571–583, 2007. ISSN 0164-1212. Software Performance.

CAPGEMINI. **World Quality Report 2024-25**. 2025. <https://www.capgemini.com/insights/research-library/world-quality-report-2024-25/>. (Accessed on 28/09/2025).

CERVANTES, A. Exploring the use of a test automation framework. In: **2009 IEEE Aerospace conference**. [S.l.: s.n.], 2009. p. 1–9.

CHEN, D. et al. Llm for mobile: An initial roadmap. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 34, n. 5, maio 2025. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/3708528>.

CHEN, H. et al. Agenttester: An llm-based tool for unit test generation with automatically generated prompts. In: HUANG, D.-S. et al. (Ed.). **Advanced Intelligent Computing Technology and Applications**. Singapore: Springer Nature Singapore, 2025. p. 114–126. ISBN 978-981-95-0011-6.

CHOUDHARY, S. R.; GORLA, A.; ORSO, A. Automated test input generation for android: Are we there yet? (e). In: **2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2015. p. 429–440.

CHUDLEIGH, M. F. et al. A guideline for hazop studies on systems which include a programmable electronic system. In: RABE, G. (Ed.). **Safe Comp 95**. London: Springer London, 1995. p. 42–58. ISBN 978-1-4471-3054-3.

COLLINS, E. et al. Deep reinforcement learning based android application gui testing. In: **Proceedings of the XXXV Brazilian Symposium on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2021. (SBES'21), p. 186–194. ISBN 9781450390613.

CORPORATION, M. **Visual Studio App Center**. 2023. <https://visualstudio.microsoft.com/app-center/>. Last accessed: February 2023.

CORPORATION, M. **Visual Studio App Center**. 2024. <https://visualstudio.microsoft.com/app-center/>. Last accessed: February 2024.

DELAMARO, M.; MALDONADO, J.; MATHUR, A. Interface mutation: an approach for integration testing. **IEEE Transactions on Software Engineering**, v. 27, n. 3, p. 228–247, 2001.

DELAMARO, M.; PEZZè, M.; VINCENZI, A. Mutant operators for testing concurrent java programs. In: **Anais do XV Simpósio Brasileiro de Engenharia de Software**. Porto Alegre, RS, Brasil: SBC, 2001. p. 272–285. ISSN 2833-0633. Disponível em: <https://sol.sbc.org.br/index.php/sbes/article/view/23994>.

DEMILLO, R.; LIPTON, R.; SAYWARD, F. Hints on test data selection: Help for the practicing programmer. **Computer**, v. 11, n. 4, p. 34–41, 1978.

DENG, L. et al. Mutation operators for testing Android apps. **Information and Software Technology**, v. 81, p. 154–168, jan. 2017. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584916300684>.

DENG, L.; OFFUTT, J.; SAMUDIO, D. Is mutation analysis effective at testing android apps? In: **2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)**. Prague, Czech Republic: IEEE, 2017. p. 86–93.

DEREZIńSKA, A.; HAłAS, K. Analysis of Mutation Operators for the Python Language. In: ZAMOJSKI, W. et al. (Ed.). **Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30 – July 4, 2014, Brunów, Poland**. Cham: Springer International Publishing, 2014. p. 155–164. ISBN 978-3-319-07013-1.

DEURSEN, A. et al. **Refactoring test code**. NLD, 2001.

DHANAPAL, K. B. et al. An innovative system for remote and automated testing of mobile phone applications. In: **2012 Annual SRII Global Conference**. [S.l.: s.n.], 2012. p. 44–54.

DINH, H. T. et al. A survey of mobile cloud computing: architecture, applications, and approaches. **Wireless communications and mobile computing**, Wiley Online Library, v. 13, n. 18, p. 1587–1611, 2013.

DURELLI, V. H. S. et al. Machine learning applied to software testing: A systematic mapping study. **IEEE Transactions on Reliability**, v. 68, n. 3, p. 1189–1212, 2019.

DYBå, T.; DINGSøYR, T. Empirical studies of agile software development: A systematic review. **Information and Software Technology**, v. 50, n. 9, p. 833–859, 2008. ISSN 0950-5849.

ELBAUM, S.; GABLE, D.; ROTHERMEL, G. The impact of software evolution on code coverage information. In: **Proceedings IEEE International Conference on Software Maintenance. ICSM 2001**. [S.l.: s.n.], 2001. p. 170–179.

ESCOBAR-VELáSQUEZ, C. et al. Enabling Mutant Generation for Open- and Closed-Source Android Apps. **IEEE Transactions on Software Engineering**, v. 48, n. 1, p. 186–208, jan. 2022. ISSN 1939-3520. Disponível em: <https://ieeexplore.ieee.org/document/9052435>.

EVTIKHIEV, M. et al. Out of the bleu: How should we assess quality of the code generation models? **Journal of Systems and Software**, v. 203, p. 111741, 2023. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S016412122300136X>.

FABBRI, S. et al. Improvements in the start tool to better support the systematic review process. In: **Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2016. (EASE '16), p. 5. ISBN 9781450336918.

FABBRI, S. C. P. F. et al. Externalising tacit knowledge of the systematic review process. **IET Software**, v. 7, n. 6, p. 298–307, 2013.

FARIA, K. A. C. **Uma solução baseada em economia collaborative para escalar o tested de aplicações Android em dispositivos reais**. Tese (Doutorado) — INF/UFG, Goiânia-GO, abr. 2019.

FARIA, K. A. C. et al. On using collaborative economy for test cost reduction in high fragmented environments. **Future Generation Computer Systems**, v. 95, p. 502–510, 2019. ISSN 0167-739X.

FAZZINI, M.; ORSO, A. Managing app testing device clouds: Issues and opportunities. In: **Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2021. (ASE '20), p. 1257–1259. ISBN 9781450367684. Disponível em: <https://doi.org/10.1145/3324884.3418909>.

FELIZARDO, K. R. et al. Using forward snowballing to update systematic reviews in software engineering. In: **Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement**. New York, NY, USA: Association for Computing Machinery, 2016. (ESEM '16), p. 6. ISBN 9781450344272.

Felizardo, Katia Romero et al. Evaluating strategies for forward snowballing application to support secondary studies updates: Emergent results. In: **Proceedings of the XXXII Brazilian Symposium on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2018. (SBES '18), p. 184–189. ISBN 9781450365031.

FRASER, G.; ARCURI, A. Evosuite: automatic test suite generation for object-oriented software. In: **Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2011. (ESEC/FSE '11), p. 416–419. ISBN 9781450304436. Disponível em: <https://doi.org/10.1145/2025113.2025179>.

FüRST, J. et al. Evaluating bluetooth low energy for iot. In: **2018 IEEE Workshop on Benchmarking Cyber-Physical Networks and Systems (CPSBench)**. Porto, Portugal: IEEE, 2018. p. 1–6.

GAO, J. et al. Mobile application testing: A tutorial. **Computer**, IEEE Computer Society Press, Washington, DC, USA, v. 47, n. 2, p. 46–55, fev. 2014. ISSN 0018-9162.

GARCÍA, B. et al. Use of chatgpt as an assistant in the end-to-end test script generation for android apps. In: **Proceedings of the 15th ACM International Workshop on Automating Test Case Design, Selection and Evaluation**. New York, NY, USA: Association for Computing Machinery, 2024. (A-TEST 2024), p. 5–11. ISBN 9798400711091. Disponível em: <https://doi.org/10.1145/3678719.3685691>.

GAROUSI, V.; KUCUK, B.; FELDERER, M. What we know about smells in software test code. **IEEE Software**, v. 36, n. 3, p. 61–73, 2019.

GHOBAKHLOO, M. Industry 4.0, digitization, and opportunities for sustainability. **Journal of Cleaner Production**, v. 252, p. 119869, 2020. ISSN 0959-6526.

GIRARDON, G. et al. Testing as a service (taas): a systematic literature map. In: **Proceedings of the 35th Annual ACM Symposium on Applied Computing**. New York, NY, USA: Association for Computing Machinery, 2020. (SAC '20), p. 1989–1996. ISBN 9781450368667.

GOPINATH, R.; JENSEN, C.; GROCE, A. Code coverage for suite evaluation by developers. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 72–82. ISBN 9781450327565. Disponível em: <https://doi.org/10.1145/2568225.2568278>.

GOWRISHANKAR, S.; MADHU, N.; BASAVARAJU, T. G. Role of ble in proximity based automation of iot: A practical approach. In: **2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS)**. Trivandrum, India: IEEE, 2015. p. 400–405.

GUO, C. et al. Fet: Hybrid cloud-based mobile bank application testing. In: **2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)**. [S.l.: s.n.], 2018. p. 21–26.

HAAS, R. et al. How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies. In: **Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2021. (ESEC/FSE 2021), p. 1281–1291. ISBN 9781450385626. Disponível em: <https://doi.org/10.1145/3468264.3473922>.

HALLER, K. Mobile testing. **SIGSOFT Softw. Eng. Notes**, Association for Computing Machinery, New York, NY, USA, v. 38, n. 6, p. 1–8, nov 2013. ISSN 0163-5948. Disponível em: <https://doi.org/10.1145/2532780.2532813>.

HAMZA, Z. A.; HAMMAD, M. Web and mobile applications testing using black and white box approaches. **IET Conference Publications**, Bahrain, Bahrain, v. 2019, n. CP758, 2019. Current testing;Mobile applications;Software applications;Software developer;White box;.

HARMAN, M. et al. Mutation-guided llm-based test generation at meta. In: _____. **Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2025. p. 180–191. ISBN 9798400712760. Disponível em: <https://doi.org/10.1145/3696630.3728544>.

HARROLD, M. J. Testing: A roadmap. In: **Proceedings of the Conference on The Future of Software Engineering**. New York, NY, USA: ACM, 2000. (ICSE '00), p. 61–72. ISBN 1-58113-253-0.

HEMMATI, H. How effective are code coverage criteria? In: **2015 IEEE International Conference on Software Quality, Reliability and Security**. [S.l.: s.n.], 2015. p. 151–156.

HERLIM, R. S. et al. Empirical study of effectiveness of evosuite on the sbst 2020 tool competition benchmark. In: O'REILLY, U.-M.; DEVROEY, X. (Ed.). **Search-Based Software Engineering**. Cham: Springer International Publishing, 2021. p. 121–135. ISBN 978-3-030-88106-1.

HODA, R. et al. Systematic literature reviews in agile software development: A tertiary study. **Information and Software Technology**, v. 85, p. 60–70, 2017. ISSN 0950-5849.

HOLL, K.; ELBERZHAGER, F. Quality assurance of mobile applications: A systematic mapping study. In: **Proceedings of the 15th International Conference on Mobile and Ubiquitous Multimedia**. New York, NY, USA: Association for Computing Machinery, 2016. (MUM '16), p. 101–113. ISBN 9781450348607.

HUANG, J.-f. Appacts: Mobile app automated compatibility testing service. In: **2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering**. [S.l.: s.n.], 2014. p. 85–90.

INOZEMTSEVA, L.; HOLMES, R. Coverage is not strongly correlated with test suite effectiveness. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 435–445. ISBN 9781450327565.

ISLAM, R.; ISLAM, R.; MAZUMDER, T. Mobile application and its global impact. **International Journal of Engineering & Technology**, Citeseer, v. 10, n. 6, p. 72–78, 2010.

ISO/IEC 25010. **ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models**. 2011.

ISO/IEC/IEEE 29119-1. Iso/iec/ieee international standard - software and systems engineering –software testing –part 1:general concepts. **ISO/IEC/IEEE 29119-1:2022(E)**, p. 1–60, 2022.

IVARSSON, M.; GORSCHEK, T. A method for evaluating rigor and industrial relevance of technology evaluations. **Empirical Softw. Engg.**, Kluwer Academic Publishers, USA, v. 16, n. 3, p. 365–395, jun 2011. ISSN 1382-3256.

JABBARVAND, R.; MALEK, S. µDroid: an energy-aware mutation testing framework for Android. In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 208–219. ISBN 978-1-4503-5105-8. Disponível em: <https://dl.acm.org/doi/10.1145/3106237.3106244>.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. **IEEE Transactions on Software Engineering**, v. 37, n. 5, p. 649–678, 2011.

JOORABCHI, M. E.; MESBAH, A.; KRUCHTEN, P. Real challenges in mobile app development. In: **2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2013. p. 15–24.

JUN, W.; MENG, F. Software testing based on cloud computing. In: **2011 International Conference on Internet Computing and Information Services**. [S.l.: s.n.], 2011. p. 176–178.

JúNIOR, M. C. et al. Dynamic testing techniques of non-functional requirements in mobile apps: A systematic mapping study. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, dec 2021. ISSN 0360-0300. Just Accepted.

JUST, R.; KURTZ, B.; AMMANN, P. Inferring mutant utility from program context. In: **Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis**. New York, NY, USA: Association for Computing Machinery, 2017. (ISSTA 2017), p. 284–294. ISBN 9781450350761. Disponível em: <https://doi.org/10.1145/3092703.3092732>.

KAASILA, J. et al. Testdroid: Automated remote ui testing on android. In: **Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia**. New York, NY, USA: Association for Computing Machinery, 2012. (MUM '12), p. 1–4. ISBN 9781450318150.

KARHU, K. et al. Empirical observations on software testing automation. In: **2009 International Conference on Software Testing Verification and Validation**. [S.l.: s.n.], 2009. p. 201–209.

KASNECI, E. et al. Chatgpt for good? on opportunities and challenges of large language models for education. **Learning and Individual Differences**, v. 103, p. 102274, 2023. ISSN 1041-6080. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1041608023000195>.

KHAN, M. K.; BRYCE, R. Android gui test generation with sarsa. In: **2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)**. [S.l.: s.n.], 2022. p. 0487–0493.

KHAN, M. N. A. et al. A literature review on software testing techniques for smartphone applications. **Engineering, Technology & Applied Science Research**, v. 10, n. 6, p. 6578–6583, Dec. 2020.

KIM, S.; CLARK, J.; MCDERMID, J. Class mutation: Mutation testing for object-oriented programs. In: **Proc. Net. ObjectDays**. [S.l.: s.n.], 2000. p. 9–12.

KIM, S.; CLARK, J.; MCDERMID, J. The rigorous generation of java mutation using hazop. In: **In Proceedings of the 12 the International Conference on Software and Systems Engineering and Their Applications (ICSSEA'99)**. Paris, France: TBA, 2000.

KITCHENHAM, B. et al. Systematic literature reviews in software engineering – a tertiary study. **Information and Software Technology**, v. 52, n. 8, p. 792–805, 2010. ISSN 0950-5849.

KITCHENHAM, B. A.; CHARTERS, S. **Guidelines for performing Systematic Literature Reviews in Software Engineering**. [S.l.], 2007.

KOCHHAR, P. S.; THUNG, F.; LO, D. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In: **2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.: s.n.], 2015. p. 560–564.

KONG, P. et al. Automated testing of android apps: A systematic literature review. **IEEE Transactions on Reliability**, v. 68, n. 1, p. 45–66, March 2019. ISSN 1558-1721.

KONG, Q. et al. Prophetagent: Automatically synthesizing gui tests from test cases in natural language for mobile apps. In: _____. **Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2025. p. 174–179. ISBN 9798400712760. Disponível em: <https://doi.org/10.1145/3696630.3728543>.

KUDO, T. N.; BULCãO-NETO, R. F.; VINCENZI, A. M. Requirement patterns: a tertiary study and a research agenda. **IET Software**, Institution of Engineering and Technology, v. 14, p. 18–26(8), February 2020. ISSN 1751-8806.

KUDO, T. N. et al. Using evidence from systematic studies to guide a phd research in requirements engineering: an experience report. **Journal of Software Engineering Research and Development**, 2022. (To appear).

KULESOVS, I. ios applications testing. **Vide. Tehnologija. Resursi - Environment, Technology, Resources**, Rezekne, Latvia, v. 3, p. 138 – 150, 2015. ISSN 16915402.

KUROISHI, P. et al. Designing mutation operators for android device components: A view through bluetooth and location api's. In: **Anais do XXXIX Simpósio Brasileiro de Engenharia de Software**. Porto Alegre, RS, Brasil: SBC, 2025. p. 149–159. ISSN 2833-0633. Disponível em: <https://sol.sbc.org.br/index.php/sbes/article/view/36994>.

KUROISHI, P. H.; MALDONADO, J. C.; VINCENZI, A. M. R. Towards the implementation of a mobile application testing infrastructure at von braun labs. In: **2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.: s.n.], 2023. p. 91–101.

KUROISHI, P. H.; MALDONADO, J. C.; VINCENZI, A. M. R. Towards the definition of a research agenda on mobile application testing based on a tertiary study. **Information and Software Technology**, v. 167, p. 107363, 2024. ISSN 0950-5849.

LAKSHMI, M. et al. Customer's activity recognition in smart retail environment using altbeacon. In: SHETTY, N. R. et al. (Ed.). **Emerging Research in Computing, Information, Communication and Applications**. Singapore: Springer Singapore, 2019. p. 591–604. ISBN 978-981-13-5953-8.

LANUI, A.; CHIEW, T. K. A cloud-based solution for testing applications' compatibility and portability on fragmented android platform. In: **2019 26th Asia-Pacific Software Engineering Conference (APSEC)**. [S.l.: s.n.], 2019. p. 158–164.

LI, C. et al. Elegant: Towards effective location of fragmentation-induced compatibility issues for android apps. In: **2018 25th Asia-Pacific Software Engineering Conference (APSEC)**. [S.l.: s.n.], 2018. p. 278–287.

LIANG, C.-J. M. et al. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In: **Proceedings of the 20th Annual International Conference on Mobile Computing and Networking**. New York, NY, USA: Association for Computing Machinery, 2014. (MobiCom '14), p. 519–530. ISBN 9781450327831.

LIKERT, R. A technique for the measurement of attitudes. **Archives of psychology**, 1932.

LIN, H. et al. Virtual device farms for mobile app testing at scale: A pursuit for fidelity, efficiency, and accessibility. In: **Proceedings of the 29th Annual International Conference on Mobile Computing and Networking**. New York, NY, USA: Association for Computing Machinery, 2023. (ACM MobiCom '23), p. 1–17. ISBN 9781450399906.

LINARES-VáSQUEZ, M. et al. Enabling mutation testing for android apps. In: **Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 233–244. ISBN 9781450351058. Disponível em: <https://doi.org/10.1145/3106237.3106275>.

LIU, C.-H. A compatibility testing platform for android multimedia applications. **Multimedia Tools and Applications**, v. 78, n. 4, p. 4885–4904, Feb 2019. ISSN 1573-7721.

LIU, J. et al. Droidmutator: an effective mutation analysis tool for android applications. In: **Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings**. New York, NY, USA: Association for Computing Machinery, 2020. (ICSE '20), p. 77–80. ISBN 9781450371223. Disponível em: <https://doi.org/10.1145/3377812.3382134>.

LIU, P. et al. Automatically detecting incompatible android apis. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 33, n. 1, nov 2023. ISSN 1049-331X.

LIU, Y.; ZHANG, T.; CHENG, J. Survey on crowd-based mobile app testing. In: **Proceedings of the 2019 11th International Conference on Machine Learning and Computing**. New York, NY, USA: Association for Computing Machinery, 2019. (ICMLC '19), p. 521–527. ISBN 9781450366007. Disponível em: <https://doi.org/10.1145/3318299.3318312>.

LIU, Z. et al. **Chatting with GPT-3 for Zero-Shot Human-Like Mobile Automated GUI Testing**. 2023.

LIU, Z. et al. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In: **Proceedings of the 45th International Conference on Software Engineering**. IEEE Press, 2023. (ICSE '23), p. 1355–1367. ISBN 9781665457019. Disponível em: <https://doi.org/10.1109/ICSE48619.2023.00119>.

LIU, Z. et al. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In: **Proceedings of the IEEE/ACM 46th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2024. (ICSE '24). ISBN 9798400702174. Disponível em: <https://doi.org/10.1145/3597503.3639180>.

LIU, Z.; GAO, X.; LONG, X. Adaptive random testing of mobile application. In: **2010 2nd International Conference on Computer Engineering and Technology**. [S.l.: s.n.], 2010. v. 2, p. V2–297–V2–301.

LLC, G. **Firebase Test Lab**. 2023. <https://firebase.google.com/docs/test-lab>. Last accessed: February 2023.

LLC, G. **Firebase Test Lab**. 2024. <https://firebase.google.com/docs/test-lab>. Last accessed: February 2024.

LLC., G. **Bluetooth overview**. 2025. <https://developer.android.com/develop/connectivity/bluetooth/>. (Accessed on 28/03/2025).

LLC., G. **Location overview**. 2025. <https://developer.android.com/develop/sensors-and-location/location>. (Accessed on 28/03/2025).

LLC., G. **Overview of Google Play Services**. 2025. <https://developers.google.com/android/guides/overview>. (Accessed on 28/03/2025).

LUNA, E.; ARISS, O. E. Edroid: A mutation tool for android apps. In: **2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)**. San Luis Potosi, Mexico: IEEE, 2018. p. 99–108.

LUO, C. et al. A survey of context simulation for testing mobile context-aware applications. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 53, n. 1, fev. 2020. ISSN 0360-0300.

LUO, C. et al. Camtest: A laboratory testbed for camera-based mobile sensing applications. **Pervasive and Mobile Computing**, v. 56, p. 106–131, 2019. ISSN 1574-1192.

LUO, L. Software testing techniques. **Institute for Software Research International Carnegie Mellon University Pittsburgh, PA**, v. 15232, n. 1-19, p. 19, 2001.

MA, L. et al. Deepmutation: Mutation testing of deep learning systems. In: **2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)**. Memphis, TN, USA: IEEE, 2018. p. 100–111.

MA, X. et al. An automated testing platform for mobile applications. In: **2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)**. [S.l.: s.n.], 2016. p. 159–162.

MA, Y.-S.; KWON, Y.-R.; OFFUTT, J. Inter-class mutation operators for java. In: **13th International Symposium on Software Reliability Engineering, 2002. Proceedings.** Annapolis, MD, USA: IEEE, 2002. p. 352–363.

MA, Y.-S.; OFFUTT, J.; KWON, Y.-R. MuJava: a mutation system for java. In: **Proceedings of the 28th international conference on Software engineering**. New York, NY, USA: Association for Computing Machinery, 2006. (ICSE '06), p. 827–830. ISBN 978-1-59593-375-1. Disponível em: <https://doi.org/10.1145/1134285.1134425>.

MALINI, A. et al. Mobile application testing on smart devices using mtaas framework in cloud. In: **International Conference on Computing and Communication Technologies**. [S.l.: s.n.], 2014. p. 1–5.

MARATHONLABS. **Marathon**. 2023. <https://docs.marathonlabs.io/>. Last accessed: February 2023.

MATYI, H. et al. Digitalization in industry 4.0: The role of mobile devices. **Journal of Production Engineering**, v. 23, n. 1, p. 75–78, 2020.

MENEGASSI, A. A.; ENDO, A. T. An evaluation of automated tests for hybrid mobile applications. In: **2016 XLII Latin American Computing Conference (CLEI)**. [S.l.: s.n.], 2016. p. 1–11.

MILTENBERGER, M. et al. Dfarm: Massive-scaling dynamic android app analysis on real hardware. In: **2020 IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft)**. [S.l.: s.n.], 2020. p. 12–15.

MIRSHOKRAIE, S.; MESBAH, A.; PATTABIRAMAN, K. Efficient javascript mutation testing. In: **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation**. Luxembourg, Luxembourg: IEEE, 2013. p. 74–83.

MOREIRA, J. S.; ALVES, E. L. G.; ANDRADE, W. L. A systematic mapping on energy efficiency testing in android applications. In: **19th Brazilian Symposium on Software Quality**. New York, NY, USA: Association for Computing Machinery, 2020. (SBQS'20), p. 10. ISBN 9781450389235.

MOURãO, E. et al. On the performance of hybrid search strategies for systematic literature reviews in software engineering. **Information and Software Technology**, v. 123, p. 106294, 2020. ISSN 0950-5849.

MOZGOVOY, M.; PYSHKIN, E. Mobile farm for software testing. In: **Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct**. New York, NY, USA: Association for Computing Machinery, 2018. (MobileHCI '18), p. 31–38. ISBN 9781450359412.

MUCCINI, H.; FRANCESCO, A. D.; ESPOSITO, P. Software testing of mobile applications: Challenges and future research directions. In: **2012 7th International Workshop on Automation of Software Test (AST)**. [S.l.: s.n.], 2012. p. 29–35.

MUSHROOR, S.; HAQUE, S.; AMIR, R. A. The impact of smart phones and mobile devices on human health and life. **International Journal Of Community Medicine And Public Health**, v. 7, n. 1, p. 9–15, Dec. 2019.

MUZAMAL, M.; NADEEM, A. Improving test adequacy assessment by novel javascript mutation operators. In: **2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)**. Islamabad, Pakistan: IEEE, 2019. p. 647–652.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. [S.l.]: Wiley Publishing, 2011. ISBN 1118031962, 9781118031964.

MéNDEZ-PORRAS, A.; QUESADA-LóPEZ, C.; JENKINS, M. Automated testing of mobile applications: A systematic map and review. **CIBSE 2015 - XVIII Ibero-American Conference on Software Engineering**, p. 195–208, 2015.

NAN, Z. et al. Test Intention Guided LLM-Based Unit Test Generation . In: **2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)**. Los Alamitos, CA, USA: IEEE Computer Society, 2025. p. 1026–1038. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00243>.

NETWORK, R.; YOUNG, D. G. **AltBeacon overview**. 2025. <https://altbeacon.github.io/android-beacon-library/>. (Accessed on 28/03/2025).

NIE, L. et al. A systematic mapping study for graphical user interface testing on mobile apps. **IET Software**, n/a, n. n/a, 2023.

NISHIURA, K. et al. Mutation analysis for javascriptweb application testing. In: **SEKE**. Boston, Massachusetts, USA: KSI Research Inc, 2013. v. 2013, p. 159–165.

OUEDRAOGO, W. C. et al. Llms and prompting for unit test generation: A large-scale evaluation. In: **Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2024. (ASE '24), p. 2464–2465. ISBN 9798400712487. Disponível em: <https://doi.org/10.1145/3691620.3695330>.

PAIVA, A. C. R. et al. Testing when mobile apps go to background and come back to foreground. In: **2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. Xi'an, China: IEEE, 2019. p. 102–111.

PAN, M. et al. Reinforcement learning based curiosity-driven testing of android applications. In: **Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis**. New York, NY, USA: Association for Computing Machinery, 2020. (ISSTA 2020), p. 153–164. ISBN 9781450380089. Disponível em: <https://doi.org/10.1145/3395363.3397354>.

PAPADAKIS, M. et al. Chapter six - mutation testing advances: An analysis and survey. In: MEMON, A. M. (Ed.). **Advances in Computers**. Elsevier, 2019, (Advances in Computers, v. 112). p. 275–378. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0065245818300305>.

PAPINENI, K. et al. Bleu: a method for automatic evaluation of machine translation. In: **Proceedings of the 40th Annual Meeting on Association for Computational Linguistics**. USA: Association for Computational Linguistics, 2002. (ACL '02), p. 311–318. Disponível em: <https://doi.org/10.3115/1073083.1073135>.

PECORELLI, F. et al. Testing of mobile applications in the wild: A large-scale empirical study on android apps. In: **Proceedings of the 28th International Conference on Program Comprehension**. New York, NY, USA: Association for Computing Machinery, 2020. (ICPC '20), p. 296–307. ISBN 9781450379588.

PECORELLI, F. et al. Software testing and android applications: a large-scale empirical study. **Empirical Software Engineering**, v. 27, n. 2, p. 31, Dec 2021. ISSN 1573-7616.

PETERSEN, K. et al. Systematic mapping studies in software engineering. In: **Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering**. Swindon, GBR: BCS Learning & Development Ltd., 2008. (EASE'08), p. 68–77.

PETERSEN, K.; VAKKALANKA, S.; KURNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering. **Inf. Softw. Technol.**, Butterworth-Heinemann, USA, v. 64, n. C, p. 1–18, ago. 2015. ISSN 0950-5849.

PETROVIć, G. et al. An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions. In: **2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.: s.n.], 2018. p. 47–53.

PETROVIć, G.; IVANKOVIć, M. State of mutation testing at Google. In: **Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice**. New York, NY, USA: Association for Computing Machinery, 2018. (ICSE-SEIP'18, v. 1), p. 163–171.

PHUNG, H. et al. Visidroid: An approach for generating test scripts from task descriptions for mobile testing. In: HADFI, R. et al. (Ed.). **PRICAI 2024: Trends in Artificial Intelligence**. Singapore: Springer Nature Singapore, 2025. p. 67–78. ISBN 978-981-96-0128-8.

PIZZOLETO, A. V. et al. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. **Journal of Systems and Software**, v. 157, p. 110388, 2019. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121219301554>.

POLO-USAOLA, M.; RODRíGUEZ-TRUJILLO, I. Analysing the combination of cost reduction techniques in android mutation testing. **Software Testing, Verification and Reliability**, v. 31, n. 7, p. e1769, 2021. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1769>.

POPOVIĆ, M. chrF: character n-gram F-score for automatic MT evaluation. In: BOJAR, O. et al. (Ed.). **Proceedings of the Tenth Workshop on Statistical Machine Translation**. Lisbon, Portugal: Association for Computational Linguistics, 2015. p. 392–395. Disponível em: <https://aclanthology.org/W15-3049/>.

POTUZAK, T.; LIPKA, R. Current trends in automated test case generation. In: **2023 18th Conference on Computer Science and Intelligence Systems (FedCSIS)**. [S.l.: s.n.], 2023. p. 627–636.

PRATHIBHAN, C. et al. An automated testing framework for testing android mobile applications in the cloud. In: **2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies**. [S.l.: s.n.], 2014. p. 1216–1219.

RAIAAN, M. A. K. et al. A review on large language models: Architectures, applications, taxonomies, open issues and challenges. **IEEE Access**, v. 12, p. 26839–26874, 2024.

RAMLER, R.; KLAMMER, C.; BUCHGEHER, G. Applying automated test case generation in industry: A retrospective. In: **2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.: s.n.], 2018. p. 364–369.

RAMLER, R.; WOLFMAIER, K. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In: **Proceedings of the 2006 International Workshop on Automation of Software Test**. New York, NY, USA: Association for Computing Machinery, 2006. (AST '06), p. 85–91. ISBN 1595934081. Disponível em: <https://doi.org/10.1145/1138929.1138946>.

RAZALI, N. M.; YAP, B. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. **J. Stat. Model. Analytics**, v. 2, 01 2011.

REN, S. et al. **CodeBLEU: a Method for Automatic Evaluation of Code Synthesis**. 2020. Disponível em: <https://arxiv.org/abs/2009.10297>.

ROJAS, I. K. V.; MEIRELES, S.; DIAS-NETO, A. C. Cloud-based mobile app testing framework: Architecture, implementation and execution. In: **Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing**. New York, NY, USA: Association for Computing Machinery, 2016. (SAST '16), p. 1–10. ISBN 9781450347662.

SAHINOGLU, M.; INCKI, K.; AKTAS, M. S. Mobile application verification: A systematic mapping study. In: GERVASI, O. et al. (Ed.). **Computational Science and Its Applications – ICCSA 2015**. Cham: Springer International Publishing, 2015. p. 147–163. ISBN 978-3-319-21413-9.

SALLOU, J.; DURIEUX, T.; PANICHELLA, A. Breaking the silence: the threats of using llms in software engineering. In: **Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results**. New York, NY, USA: Association for Computing Machinery, 2024. (ICSE-NIER'24), p. 102–106. ISBN 9798400705007. Disponível em: <https://doi.org/10.1145/3639476.3639764>.

SANTOS, I.; FILHO, J. C. C.; SOUZA, S. R. S. A survey on the practices of mobile application testing. In: **2020 XLVI Latin American Computing Conference (CLEI)**. [S.l.: s.n.], 2020. p. 232–241.

SARWAR, M.; SOOMRO, T. Impact of smartphone's on society. **European Journal of Scientific Research**, v. 98, 02 2013.

SAVARIMUTHU, S.; WINIKOFF, M. Mutation operators for the goal agent language. In: COSSENTINO, M.; SEGHROUCHNI, A. E. F.; WINIKOFF, M. (Ed.). **Engineering Multi-Agent Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 255–273. ISBN 978-3-642-45343-4.

SCHWEIKL, S.; FRASER, G.; ARCURI, A. Evosuite at the sbst 2022 tool competition. In: **Proceedings of the 15th Workshop on Search-Based Software Testing**. New York, NY, USA: Association for Computing Machinery, 2023. (SBST '22), p. 33–34. ISBN 9781450393188. Disponível em: <https://doi.org/10.1145/3526072.3527526>.

SIDDIQ, M. L. et al. Using large language models to generate junit tests: An empirical study. In: **Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2024. (EASE '24), p. 313–322. ISBN 9798400717017. Disponível em: <https://doi.org/10.1145/3661167.3661216>.

SILVA, H. N. et al. A mapping study on mutation testing for mobile applications. **Software Testing, Verification and Reliability**, v. 32, n. 8, p. e1801, 2022. ISSN 1099-1689. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1801>.

SQUARE. **Spoon**. 2023. <http://square.github.io/spoon/>. Last accessed: February 2023.

STAROV, O.; VILKOMIR, S. Integrated taas platform for mobile development: Architecture solutions. In: **2013 8th International Workshop on Automation of Software Test (AST)**. [S.l.: s.n.], 2013. p. 1–7.

STAROV, O. et al. Testing-as-a-service for mobile applications: State-of-the-art survey. In: ZAMOJSKI, W.; SUGIER, J. (Ed.). **Dependability Problems of Complex Information Systems**. Cham: Springer International Publishing, 2015. p. 55–71. ISBN 978-3-319-08964-5.

STATCOUNTER. **Mobile Android Version Market Share Worldwide**. 2022. Available at: <https://gs.statcounter.com/android-version-market-share/mobile/worldwide/monthly-202109-202209-bar>. Accessed: 2023-03-13.

STATCOUNTER. **Mobile Operating System Market Share Worldwide**. 2022. Available at: <https://gs.statcounter.com/os-market-share/mobile/worldwide/monthly-202109-202209-bar>. Accessed: 2023-03-13.

STATISTA. **Forecast number of mobile devices worldwide from 2020 to 2025 (in billions)**. 2022. Available at: <https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide/>. Accessed: 2023-03-13.

STATISTA. **Number of apps available in leading app stores as of 3rd quarter 2022**. 2022. Available at: <https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/>. Accessed: 2023-03-13.

STATISTA. **Forecast number of mobile devices worldwide from 2020 to 2025 (in billions)**. 2023. Available at: <https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide>. Accessed: 2024-02-08.

STATISTA. **Number of available applications in the Google Play Store from December 2009 to December 2023**. 2023. Available at: <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>. Accessed: 2024-02-08.

STATISTA. **Number of smartphones sold to end users worldwide from 2007 to 2023.** 2023. Available at: <https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/>. Accessed: 2024-02-08.

SUN, X. et al. Taming android fragmentation through lightweight crowdsourced testing. **IEEE Transactions on Software Engineering**, v. 49, n. 6, p. 3599–3615, 2023.

TAIPALE, O. et al. Trade-off between automated and manual software testing. **International Journal of System Assurance Engineering and Management**, v. 2, n. 2, p. 114–125, Jun 2011. ISSN 0976-4348. Disponível em: <https://doi.org/10.1007/s13198-011-0065-6>.

TAO, C.; GAO, J. On building a cloud-based mobile testing infrastructure service system. **Journal of Systems and Software**, v. 124, p. 39–55, 2017. ISSN 0164-1212.

TAO, D.; LIN, Z.; LU, C. Cloud platform based automated security testing system for mobile internet. **Tsinghua Science and Technology**, v. 20, n. 6, p. 537–544, 2015.

THORVE, S.; SRESHTHA, C.; MENG, N. An empirical study of flaky tests in android apps. In: **2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2018. p. 534–538.

TRAMONTANA, P. et al. Automated functional testing of mobile applications: A systematic mapping study. **Software Quality Journal**, Kluwer Academic Publishers, USA, v. 27, n. 1, p. 149–201, mar. 2019. ISSN 0963-9314.

UNTCH, R. H.; OFFUTT, A. J.; HARROLD, M. J. Mutation analysis using mutant schemata. **SIGSOFT Softw. Eng. Notes**, Association for Computing Machinery, New York, NY, USA, v. 18, n. 3, p. 139–148, jul. 1993. ISSN 0163-5948. Disponível em: <https://doi.org/10.1145/174146.154265>.

VAJAK, D. et al. Environment for automated functional testing of mobile applications. In: **2018 International Conference on Smart Systems and Technologies (SST)**. [S.l.: s.n.], 2018. p. 125–130.

VASWANI, A. et al. Attention is all you need. In: **Proceedings of the 31st International Conference on Neural Information Processing Systems**. Red Hook, NY, USA: Curran Associates Inc., 2017. (NIPS'17), p. 6000–6010. ISBN 9781510860964.

VEGA, J. J. O. et al. Model-based mutation operators for timed systems: A taxonomy and research agenda. In: **2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)**. Lisbon, Portugal: IEEE, 2018. p. 325–332.

VIGANò, E. et al. Data-Driven Mutation Analysis for Cyber-Physical Systems. **IEEE Transactions on Software Engineering**, v. 49, n. 4, p. 2182–2201, abr. 2023. ISSN 1939-3520. Conference Name: IEEE Transactions on Software Engineering. Disponível em: <https://ieeexplore.ieee.org/document/9914679>.

VILLANES, I. K.; ENDO, A. T.; DIAS-NETO, A. C. A multivocal literature mapping on mobile compatibility testing. **International Journal of Computer Applications in Technology**, v. 69, n. 2, p. 173–192, 2022.

VINCENZI, A. M. R. et al. METFORD – Mutation tEsTing Framework fOR anDroid. **Journal of Systems and Software**, v. 222, p. 112332, abr. 2025. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121224003765>.

VOGL, S. et al. Evosuite at the sbst 2021 tool competition. In: **2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)**. [S.l.: s.n.], 2021. p. 28–29.

WANG, C. et al. Llmdroid: Enhancing automated mobile app gui testing coverage with large language model guidance. **Proc. ACM Softw. Eng.**, Association for Computing Machinery, New York, NY, USA, v. 2, n. FSE, jun. 2025. Disponível em: <https://doi.org/10.1145/3715763>.

WANG, J. et al. Software testing with large language models: Survey, landscape, and vision. **IEEE Trans. Softw. Eng.**, IEEE Press, v. 50, n. 4, p. 911–936, abr. 2024. ISSN 0098-5589. Disponível em: <https://doi.org/10.1109/TSE.2024.3368208>.

WANG, Q. et al. A roadmap for software testing in open-collaborative and ai-powered era. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 34, n. 5, maio 2025. ISSN 1049-331X. Disponível em: <https://doi.org/10.1145/3709355>.

WANG, S. et al. Automatic unit test generation for machine learning libraries: How far are we? In: **2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2021. p. 1548–1560.

WANG, W. et al. An empirical study of android test generation tools in industrial cases. In: **Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2018. (ASE '18), p. 738–748. ISBN 9781450359375.

WANG, Y. et al. Liredroid: Llm-enhanced test case generation for static sensitive behavior replication. In: **Proceedings of the 15th Asia-Pacific Symposium on Internetware**. New York, NY, USA: Association for Computing Machinery, 2024. (Internetware '24), p. 81–84. ISBN 9798400707056. Disponível em: <https://doi.org/10.1145/3671016.3671404>.

WANG, Y. et al. Test automation maturity improves product quality—quantitative study of open source projects using continuous integration. **Journal of Systems and Software**, v. 188, p. 111259, 2022. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121222000280>.

WEI, L.; LIU, Y.; CHEUNG, S.-C. Taming android fragmentation: characterizing and detecting compatibility issues for android apps. In: **Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2016. (ASE '16), p. 226–237. ISBN 9781450338455. Disponível em: <https://doi.org/10.1145/2970276.2970312>.

WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: **Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (EASE '14), p. 10. ISBN 9781450324762.

WU, G. et al. Appcheck: A crowdsourced testing service for android applications. In: **2017 IEEE International Conference on Web Services (ICWS)**. [S.l.: s.n.], 2017. p. 253–260.

WYNN-WILLIAMS, S. et al. Can generative ai produce test cases? an experience from the automotive domain. In: **Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2025. (FSE Companion '25), p. 456–467. ISBN 9798400712760. Disponível em: <https://doi.org/10.1145/3696630.3728568>.

XAVIER, M. G. et al. Mobile application testing on clouds: Challenges, opportunities and architectural elements. In: **2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)**. [S.l.: s.n.], 2017. p. 181–185.

YANG, L. et al. On the evaluation of large language models in unit test generation. In: **Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2024. (ASE '24), p. 1607–1619. ISBN 9798400712487. Disponível em: <https://doi.org/10.1145/3691620.3695529>.

YANG, L. et al. Quality assessment in systematic literature reviews: A software engineering perspective. **Information and Software Technology**, v. 130, p. 106397, 2021. ISSN 0950-5849.

YAÚ, B. et al. A systematic mapping study on cloud-based mobile application testing. **Journal of Information and Communication Technology**, v. 18, n. 4, p. 485–527, 2019. ISSN 1675414X.

YI, G. et al. Exploring the capability of chatgpt in test generation. In: **2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)**. [S.l.: s.n.], 2023. p. 72–80.

YIN, C. et al. A literature survey on smart cities. **Sci. China Inf. Sci.**, v. 58, n. 10, p. 1–18, 2015.

YOON, J. et al. Integrating llm-based text generation with dynamic context retrieval for gui testing. In: **2025 IEEE Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2025. p. 394–405.

ZEIN, S.; SALLEH, N.; GRUNDY, J. A systematic mapping study of mobile application testing techniques. **Journal of Systems and Software**, v. 117, p. 334–356, 2016. ISSN 0164-1212.

ZENG, X. et al. Automated test input generation for android: are we really there yet in an industrial case? In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2016. (FSE 2016), p. 987–992. ISBN 9781450342186. Disponível em: <https://doi.org/10.1145/2950290.2983958>.

ZHANG, D.; ADIPAT, B. Challenges, methodologies, and issues in the usability testing of mobile applications. **International Journal of Human–Computer Interaction**, Taylor & Francis, v. 18, n. 3, p. 293–308, 2005.

ZHANG, H. et al. Towards mutation analysis for use cases. In: **Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems**. New York, NY, USA: Association for Computing Machinery, 2016. (MODELS '16), p. 363–373. ISBN 9781450343213. Disponível em: <https://doi.org/10.1145/2976767.2976784>.

ZHANG, L. L. et al. Towards a contextual and scalable automated-testing service for mobile apps. In: **Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications**. New York, NY, USA: Association for Computing Machinery, 2017. (HotMobile '17), p. 97–102. ISBN 9781450349079.

ZHANG, S.; PI, B. Mobile functional test on taas environment. In: **2015 IEEE Symposium on Service-Oriented System Engineering**. [S.l.: s.n.], 2015. p. 315–320.

ZHANG, T.; GAO, J.; CHENG, J. Crowdsourced testing services for mobile apps. In: **2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)**. [S.l.: s.n.], 2017. p. 75–80.

ZHENG, Z. et al. Towards an understanding of large language models in software engineering tasks. **Empirical Software Engineering**, v. 30, n. 2, p. 50, Dec 2024. ISSN 1573-7616. Disponível em: <https://doi.org/10.1007/s10664-024-10602-0>.

Appendix

APPENDIX A

Supplementary information of paper P1.

A.1 List of search string applied on each database

A.1 presents the search string applied on each online repository: IEEE, ACM, Scopus, ScienceDirect, SpringerLink, ISI Web of Science, Engineering Village, and Wiley.

Table 61 – Search string applied on each database.

Database	**Search String**
IEEE	(“mobile testing” OR “mobile application testing” OR “mobile applications testing” OR “mobile app testing” OR “mobile apps testing” OR “android application testing” OR “android applications testing” OR “android app testing” OR “android apps testing” OR “ios application testing” OR “ios applications testing” OR “ios app testing” OR “ios apps testing”) AND (“systematic review” OR “systematic literature review” OR “systematic mapping” OR “systematic literature mapping” OR “mapping study” OR “literature review” OR survey)
ACM	(“mobile testing” OR “mobile application testing” OR “mobile app testing” OR “android application testing” OR “ios application testing” OR “android app testing” OR “ios app testing”) AND (“systematic literature review” OR “systematic mapping” OR “systematic literature mapping” OR “systematic review” OR “mapping study” OR “literature review” OR survey)
Scopus	TITLE-ABS-KEY((“mobile testing” OR “mobile application testing” OR “mobile app testing” OR “android application testing” OR “ios application testing” OR “android app testing” OR “ios app testing”) AND (“systematic review” OR “systematic literature review” OR “systematic mapping” OR “systematic literature mapping” OR “mapping study” OR “literature review” OR survey))

Continued on next page

Table 61 (Continued) – Search string applied on each database.

Database	Search String
ScienceDirect	(“mobile testing” AND “mobile application testing” AND “mobile app testing” AND “android application testing”) AND (“systematic literature review” OR “systematic literature mapping” OR “systematic review” OR “systematic mapping” OR survey)
SpringerLink	(“mobile testing” OR “mobile application testing” OR “mobile app testing” OR “android application testing” OR “ios application testing” OR “android app testing” OR “ios app testing”) AND (“systematic review” OR “systematic literature review” OR “systematic mapping” OR “systematic literature mapping” OR “mapping study” OR “literature review” OR survey)
ISI Web of Science	TS=((“mobile testing” OR “mobile application testing” OR “mobile app testing” OR “android application testing” OR “android app testing” OR “ios application testing” OR “ios app testing“) AND (“systematic literature review” OR “systematic literature mapping” OR “systematic review” OR “systematic mapping” OR “literature review” OR “mapping study” OR survey)) AND SU=(Computer Science)
Engineering Village	(((“mobile testing”) OR (“mobile application testing”) OR (“mobile applications testing”) OR (“mobile app testing”) OR (“mobile apps testing”) OR (“android application testing”) OR (“android applications testing”) OR (“android app testing”) OR (“android apps testing”) OR (“ios application testing”) OR (“ios applications testing”) OR (“ios app testing”) OR (“ios apps testing”)) AND ((“systematic literature review”) OR (“systematic literature mapping”) OR (“systematic review”) OR (“systematic mapping”) OR (“literature review”) OR (“mapping study”) OR (survey)))
Wiley	(“mobile testing” OR “mobile application testing” OR “mobile applications testing” OR “mobile app testing” OR “mobile apps testing” OR “android application testing” OR “android applications testing” OR “android app testing” OR “android apps testing” OR “ios application testing” OR “ios applications testing” OR “ios app testing” OR “ios apps testing”) AND (“systematic review” OR “systematic literature review” OR “systematic mapping” OR “systematic literature mapping” OR “mapping study” OR “literature review” OR survey)

A.2 List of accepted articles

A.2 present all 21 secondary studies accepted for this tertiary study.

Table 62 – List of included secondary studies.

ID	Study
S1	Sahinoglu, M., Incki, K., and Aktas, M. 2015. Mobile Application Verification: A Systematic Mapping Study. In Computational Science and Its Applications – ICCSA 2015 (pp. 147–163). Springer International Publishing.
S2	Méndez-Porras, A., Quesada-López, C., and Jenkins, M. 2015. Automated testing of mobile applications: A systematic map and review. CIBSE 2015 - XVIII Ibero-American Conference on Software Engineering, p.195-208.

Continued on next page

Table 62 (Continued) – List of included secondary studies.

ID	Study
S3	Kulesovs, I. 2015. iOS applications testing. Vide. Tehnologija. Resursi - Environment, Technology, Resources, 3, p.138 - 150.
S4	Holl, K., and Elberzhager, F. 2016. Quality Assurance of Mobile Applications: A Systematic Mapping Study. In Proceedings of the 15th International Conference on Mobile and Ubiquitous Multimedia (pp. 101–113). Association for Computing Machinery.
S5	Samer Zein, Norsaremah Salleh, and John Grundy 2016. A systematic mapping study of mobile application testing techniques. Journal of Systems and Software, 117, p.334-356.
S6	Silvia M. Ascate and Ingrid do Nascimento Mendes and Kariny Oliveira and Awdren de Lima Fontão and Isabel Karina Villanes and Arilo Dias Neto 2017. Challenges in Model-Based Testing for Mobile Applications. In Proceedings of the XX Iberoamerican Conference on Software Engineering, Buenos Aires, Argentina, May 22-23, 2017 (pp. 567–580). Curran Associates.
S7	Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T., and Klein, J. 2019. Automated Testing of Android Apps: A Systematic Literature Review. IEEE Transactions on Reliability, 68(1), p.45-66.
S8	D. R. de Almeida, P. D. L. Machado, W. L. Andrade 2019. Testing tools for Android context-aware applications: a systematic mapping. Journal of the Brazilian Computer Society, 25(1), p.12.
S9	Ya'u, B., Salleh, N., Nordin, A., Alwan, A., Idris, N., and Abas, H. 2019. A systematic mapping study on cloud-based mobile application testing. Journal of Information and Communication Technology, 18(4), p.485-527.
S10	Hamza, Z., and Hammad, M. 2019. Web and mobile applications testing using black and white box approaches. IET Conference Publications, 2019(CP758).
S11	Tramontana, P., Amalfitano, D., Amatucci, N., and Fasolino, A. 2019. Automated Functional Testing of Mobile Applications: A Systematic Mapping Study. Software Quality Journal, 27(1), p.149–201.
S12	Moreira, J., Alves, E., and Andrade, W. 2020. A Systematic Mapping on Energy Efficiency Testing in Android Applications. In 19th Brazilian Symposium on Software Quality. Association for Computing Machinery.
S13	Luo, C., Goncalves, J., Velloso, E., and Kostakos, V. 2020. A Survey of Context Simulation for Testing Mobile Context-Aware Applications. ACM Comput. Surv., 53(1).
S14	Al-Ahmad, A., Kahtan, H., Hujainah, F., and Jalab, H. 2019. Systematic Literature Review on Penetration Testing for Mobile Cloud Computing Applications. IEEE Access, 7, p.173524-173540.
S15	Misael C. Júnior, Domenico Amalfitano, Lina Garcés, Anna Rita Fasolino, Stevão A. Andrade, and Márcio Delamaro. 2021. Dynamic Testing Techniques of Non-Functional Requirements in Mobile Apps: A Systematic Mapping Study. ACM Comput. Surv. December 2021.
S16	Silva, HN, Prado Lima, J, Vergilio, SR, Endo, AT. A mapping study on mutation testing for mobile applications. Softw. Test. Verif. Reliab. 2021;e1801.
S17	Villanes, I. K., Endo, A. T. and Dias-Neto, A. C. 2022. A multivocal literature mapping on mobile compatibility testing. International Journal of Computer Applications in Technology, 69(2), pp. 173–192.
S18	L. Nie, K. S. Said, L. Ma, Y. Zheng, Y. Zhao. 2023. A systematic mapping study for graphical user interface testing on mobile apps. IET Software.

Continued on next page

Table 62 (Continued) – List of included secondary studies.

ID	Study
S19	Khan, M., Mirza, A., Wagan, R., Shahid, M., and Saleem, I. 2020. A Literature Review on Software Testing Techniques for Smartphone Applications. Engineering, Technology & Applied Science Research, 10(6), p.6578–6583.
S20	Dongsong Zhang, and Boonlit Adipat 2005. Challenges, Methodologies, and Issues in the Usability Testing of Mobile Applications. International Journal of Human–Computer Interaction, 18(3), p.293-308.
S21	Al-Ahmad, A., Aljunid, S., and Sani, A. 2013. Mobile Cloud Computing Testing Review. In 2013 International Conference on Advanced Computer Science Applications and Technologies (pp. 176-180).

A.3 List of identified gaps

A.3 display all 87 gaps and open challenges identified by each secondary study.

Table 63 – List of identified gaps and challenges.

ID	Gaps and Challenges
S1	GC1.1. Performance Testing GC1.2. Testing on Cloud GC1.3. Model-based testing
S2	GC2.1. Test case generation based on context GC2.2. Effective and efficient model creation for automation GC2.3. Testing on cloud GC2.4. Available tools for broad usage
S3	GC3.1. Lack of academic studies related to iOS application testing (automation, stress testing security testing)
S4	GC4.1. Context simulation GC4.2. Establishment of test environments GC4.3. Security-by-design GC4.4. Device-specific test coverage and fault analyses for compatibility testing GC4.5. Users review for quality assurance GC4.6. Test cases generation focusing on energy issues GC4.7. Interaction with industry to evaluate approaches GC4.8. Lack of maturity tools (automation)
S5	GC5.1. Research in real-world mobile application development environments GC5.2. Eliciting testing requirements early during the development process GC5.3. Techniques for life-cycle conformance and mobile services GC5.4. Comparative studies for security and usability testing
S6	GC6.1. Creation/improvement of existing models to address fragmentation GC.2. Validation of MBT techniques in real devices GC6.3. Difficulties to apply some models due to the specific characteristics of mobile applications GC6.4. Models to represent variations of connections, mobile platforms, and mobile contexts GC6.5. Tools to support MBT techniques

Continued on next page

Table 63 (Continued) – List of identified gaps and challenges.

ID	Gaps and Challenges
S7	GC7.1. Android fragmentation GC7.2. Concurrency testing GC7.3. Scalable tools GC7.4. Acceptance testing G7.5. Studies focusing on white-box approaches GC7.6. Regression testing techniques to identify defect-prone and unsafe updates GC7.7. Test case prioritization
S8	GC8.1. Tools to generate and execute test cases using high-level context GC8.2. Support asynchronous context variation
S9	GC9.1. Lack of general and scalable approaches to support multiple platforms and OS GC9.2. Lack of evaluation methods such as case studies to validate the proposed approaches GC9.3. Investigate other aspects of cloud-based mobile application testing (performance, regression, load, usability, interrupt, memory leakage, installation, operational, laboratory, certification, location, and outdated software testing) GC9.4. Scalable approaches to integrate mobile TaaS GC9.5. Reuse of Requirements
S10	No research gap identified
S11	GC11.1. Few contributions from industry GC11.2. iOS testing GC11.3. Testing tools based on Google Espresso GC11.4. Testing techniques aiming at C++ based components GC11.5. Context-aware studies GC11.6. Fault detection studies GC11.7. The absence of venus and journals focused on mobile testing automation
S12	GC12.1. Energy-aware prioritization test strategy GC12.2. Test case generation based on different contexts for energy efficiency GC12.3. Reduce the time for test case generation G12.4. Pinpoint wake-locks anomalies GC12.5. Improve accuracy and reduce tracing overhead when measuring energy consumption GC12.6. Time-lapse simulations GC12.7. Classify applications according to the user's typical usage GC12.8. Development of a power model to estimate the energy cost of different types of ATEBs
S13	GC13.1. Context simulation in early-stage testing GC13.2. Emulation fidelity GC13.3. Context heterogeneity GC13.4. Multi-device context-aware applications GC13.5. Automation support
S14	GC14.1. Offloading parameter is disregarded GC14.2. Vulnerabilities of mobile, cloud, and web GC14.3. MCC application penetration testing model

Continued on next page

Table 63 (Continued) – List of identified gaps and challenges.

ID	Gaps and Challenges
S15	GC15.1. Addressing specific quality characteristics GC15.2. NFRs testing of mobile hybrid apps GC15.3. Making supporting testing tools available GC15.4. Sharing industrial experience with the academic community GC15.5. Using a consolidate terminology for quality characteristics
S16	GC16.1. Specific events GC16.2. Conduct more empirical studies using mutation testing – evaluate the test suite quality and state of the art in the automated test input generation GC16.3. Target other mobile platforms, OS and programming language GC16.4. Explore the application of mutation operators in different artifacts like ASTs, bytecode, and DEX files GC16.5. Explore other kinds of operators related to context awareness and non-functional attributes (e.g., security, performance, and accessibility) GC16.6. Improve tool support for mutant execution and analysis GC16.7. Offer mechanisms to deal with stillborn, equivalent, and stubborn mutants GC16.8. Investigate strategies to reduce mutant execution costs and optimize Android mutation systems GC16.9. Offer benchmarks and conduct more rigorous studies to evaluate mutation testing for mobile apps
S17	GC17.1. A way to assist developers in updating mobile app code with the constant evolution of APIs GC17.2. Support developers to use good mobile app development practices to address compatibility issues GC17.3. Explore different app information and device characteristics to improve the selection of devices for testing
S18	GC18.1. Testing Processes and Approaches GC18.2. Test case generation GC18.3. Understanding the app behavior GC18.4. Test Minimization and Prioritization GC18.5. Graphical User Interface Testing Objectives
S19	GC19.1. Automation tool according to a selected model
S20	GC20.1. Usability testing of multimedia applications
S21	GC21.1. Lack of a specific model for MCC application testing

APPENDIX B

Supplementary information of paper P6.

B.1 Statistics of tests generated by the LLMs.

Table 64 – Statistics of Claude.

APP	CLASS	N_{TEST}	N_{IGN}	N_{EXEC}
A2DP	Connector	29	0	29
AOBD	MainActivity	37	26	11
	BtCommService	14	1	13
	BtDeviceListActivity	15	0	15
ASCL	ScanFilter	50	1	49
	PendingIntentReceiver	23	1	22
	BluetoothLeScannerImplOreo	14	4	10
	BluetoothLeScannerImplLollipop	19	2	17
	BluetoothLeScannerImplJB	29	16	13
	BluetoothLeScannerImplMarshmallow	16	0	16
AUTH	BluetoothDeviceListing	31	0	31
	HidDeviceController	6	0	6
	HidDeviceProfile	8	0	8
	BluetoothUtils	25	0	25
	HidDeviceApp	5	0	5
	BluetoothForegroundService	8	0	8
C2T	BleConnectActivity	16	1	15
	Position	19	0	19
	Periodic	13	0	13
	BleGattWrapper	26	5	21
	Smart	16	0	16

	BluetoothConnectActivity	20	6	14
GB	BleConnectActivity	16	1	15
	Position	19	0	19
	Periodic	13	0	13
	BleGattWrapper	26	5	21
	Smart	16	0	16
	BluetoothConnectActivity	20	6	14
	BondingUtil	29	3	26
	BluetoothConnectReceiver	17	1	16
	PhoneGpsLocationProvider	13	0	13
	BtBRQueue	30	1	29
	DeviceHelper	27	1	26
GPSL	Gpx10FileLogger	17	0	17
	GeneralLocationListener	16	1	15
	CustomUrlManager	24	1	23
	Kml22FileLogger	14	0	14
	GpsLoggingService	40	11	29
	Gpx11FileLogger	12	0	12
	GeoJSONLogger	14	0	14
	CSVFileLogger	20	3	17
	Locations	20	0	20
	Maths	23	0	23
	Session	33	0	33
	GpxReader	10	0	10
	GeoJSONWriterPoints	13	0	13
RU	TCX	21	4	17
	TrackerGPS	2	1	1
	GpsStatus	29	0	29
	ActivityCleaner	14	0	14
	PathSimplifier	18	0	18
ULOG	LoggerService	19	1	18
	LocationFormatter	34	0	34
	LoggerTask	11	2	9
	LocationHelper	34	6	28
WS	ReceiverService	23	11	12
	ConfigManager	17	0	17
	AirDropBleController	20	1	19
Total		1163	123	1040

Table 65 – Statistics of DeepSeek.

APP	**CLASS**	$\mathbf{N_{TEST}}$	$\mathbf{N_{IGN}}$	$\mathbf{N_{EXEC}}$
A2DP	Connector	4	1	3
AOBD	MainActivity	6	6	0
	BtCommService	4	0	4
	BtDeviceListActivity	5	1	4

ASCL	BluetoothLeScannerImplJB	6	0	6
	BluetoothLeScannerImplMarshmallow	5	0	5
	PendingIntentReceiver	4	0	4
	ScanFilter	18	0	18
	BluetoothLeScannerImplOreo	5	0	5
	BluetoothLeScannerImplLollipop	6	0	6
AUTH	BluetoothForegroundService	6	0	6
	BluetoothUtils	8	0	8
	HidDeviceApp	1	0	1
	HidDeviceController	10	1	9
	BluetoothDeviceListing	14	2	12
	HidDeviceProfile	2	0	2
C2T	BleGattWrapper	8	6	2
	BleConnectActivity	4	0	4
	Periodic	4	3	1
	Position	6	1	5
	Smart	5	5	0
	BluetoothConnectActivity	3	0	3
GB	GBLocationListener	5	0	5
	BluetoothPairingRequestReceiver	5	0	5
	PhoneNetworkLocationProvider	4	1	3
	GBDeviceCandidate	16	0	16
	BluetoothConnectReceiver	6	0	6
	BtBRQueue	4	1	3
	BluetoothStateChangeReceiver	4	0	4
	DeviceManager	6	2	4
	BondingUtil	9	3	6
	DeviceHelper	6	1	5
	PhoneGpsLocationProvider	4	0	4
GPSL	GeneralLocationListener	3	0	3
	CSVFileLogger	7	0	7
	GeoJSONLogger	7	0	7
	Kml22FileLogger	7	0	7
	GeoJSONWriterPoints	5	0	5
	GpxReader	9	0	9
	CustomUrlManager	11	1	10
	Locations	9	0	9
	Session	23	0	23
	Maths	9	0	9
	GpsLoggingService	13	0	13
	Gpx11FileLogger	3	1	2
	Gpx10FileLogger	6	1	5
RU	PathSimplifier	10	5	5
	TCX	0	0	0
	ActivityCleaner	5	4	1
	GpsStatus	4	1	3

	TrackerGPS	4	2	2
ULOG	LocationHelper	12	1	11
	LoggerTask	3	0	3
	LoggerService	7	5	2
	LocationFormatter	8	0	8
WS	AirDropBleController	5	0	5
	ConfigManager	4	0	4
	ReceiverService	5	0	5
Total		382	55	327

Table 66 – Statistics of ChatGPT.

APP	**CLASS**	$\mathbf{N_{TEST}}$	$\mathbf{N_{IGN}}$	$\mathbf{N_{EXEC}}$
A2DP	Connector	4	1	3
AOBD	BtDeviceListActivity	3	1	2
	MainActivity	8	8	0
	BtCommService	4	0	4
ASCL	PendingIntentReceiver	6	0	6
	BluetoothLeScannerImplOreo	10	3	7
	BluetoothLeScannerImplLollipop	4	0	4
	BluetoothLeScannerImplMarshmallow	4	0	4
	BluetoothLeScannerImplJB	5	2	3
	ScanFilter	15	0	15
AUTH	BluetoothDeviceListing	6	1	5
	HidDeviceController	10	0	10
	HidDeviceProfile	6	0	6
	HidDeviceApp	8	0	8
	BluetoothUtils	9	0	9
	BluetoothForegroundService	4	0	4
C2T	BluetoothConnectActivity	9	2	7
	BleConnectActivity	4	0	4
	Smart	5	0	5
	Position	7	0	7
	Periodic	3	0	3
	BleGattWrapper	6	0	6
GB	GBLocationListener	5	0	5
	BtBRQueue	4	0	4
	BluetoothStateChangeReceiver	3	0	3
	GBDeviceCandidate	6	0	6
	PhoneNetworkLocationProvider	5	0	5
	BluetoothPairingRequestReceiver	5	0	5
	BluetoothConnectReceiver	3	0	3
	PhoneGpsLocationProvider	5	1	4
	BondingUtil	5	0	5
	DeviceHelper	4	0	4
	DeviceManager	5	0	5

GPSL	GeoJSONWriterPoints	3	0	3
	GeneralLocationListener	5	3	2
	Session	19	0	19
	Maths	11	0	11
	CSVFileLogger	3	0	3
	Gpx11FileLogger	3	0	3
	Kml22FileLogger	3	0	3
	GpsLoggingService	8	0	8
	Locations	7	0	7
	GpxReader	3	0	3
	Gpx10FileLogger	4	0	4
	CustomUrlManager	5	1	4
	GeoJSONLogger	4	0	4
RU	PathSimplifier	4	0	4
	TrackerGPS	7	0	7
	ActivityCleaner	4	1	3
	GpsStatus	6	0	6
	TCX	2	0	2
ULOG	LocationFormatter	5	0	5
	LocationHelper	12	0	12
	LoggerTask	4	2	2
	LoggerService	4	0	4
WS	ConfigManager	5	0	5
	ReceiverService	6	2	4
	AirDropBleController	4	0	4
Total		331	28	303

Table 67 – Statistics of Qwen.

APP	**CLASS**	**N_{TEST}**	**N_{IGN}**	**N_{EXEC}**
A2DP	Connector	4	3	1
AOBD	BtCommService	6	3	3
	BtDeviceListActivity	5	1	4
	MainActivity	3	3	0
ASCL	BluetoothLeScannerImplOreo	5	1	4
	BluetoothLeScannerImplLollipop	6	0	6
	BluetoothLeScannerImplJB	5	0	5
	PendingIntentReceiver	4	0	4
	BluetoothLeScannerImplMarshmallow	3	0	3
	ScanFilter	28	0	28
AUTH	BluetoothForegroundService	8	5	3
	HidDeviceController	14	0	14
	BluetoothUtils	8	0	8
	HidDeviceProfile	9	0	9
	HidDeviceApp	5	1	4
	BluetoothDeviceListing	14	4	10

C2T	Periodic	4	1	3
	BleGattWrapper	8	5	3
	BluetoothConnectActivity	8	3	5
	Position	6	3	3
	BleConnectActivity	10	2	8
	Smart	6	3	3
GB	BluetoothConnectClaude	3	1	2
	GBLocationListener	6	1	5
	DeviceHelper	6	0	6
	BtBRQueue	4	1	3
	PhoneNetworkLocationProvider	1	0	1
	GBDeviceCandidate	9	0	9
	BluetoothStateChangeReceiver	4	0	4
	BluetoothPairingRequestReceiver	5	1	4
	BondingUtil	8	0	8
	PhoneGpsLocationProvider	4	2	2
	DeviceManager	4	1	3
GPSL	Gpx11FileLogger	3	2	1
	Sessions	24	1	23
	CSVFileLogger	5	0	5
	Locations	9	3	6
	Kml22FileLogger	5	0	5
	CustomUrlManager	10	3	7
	Gpx10FileLogger	8	1	7
	GeoJSONWriterPoints	2	0	2
	Maths	11	1	10
	GeoJSONLogger	3	0	3
	GpxReader	1	0	1
	GeneralLocationListener	12	0	12
	GpsLoggingService	12	11	1
RU	PathSimplifier	7	3	4
	GpsStatus	9	0	9
	TrackerGPS	5	4	1
	ActivityCleaner	4	2	2
	TCX	5	4	1
ULOG	LoggerService	5	1	4
	LocationHelper	15	2	13
	LoggerTask	6	6	0
	LocationFormatter	9	2	7
WS	ConfigManager	7	0	7
	ReceiverService	8	1	7
	AirDropBleController	4	0	4
Total		412	92	320